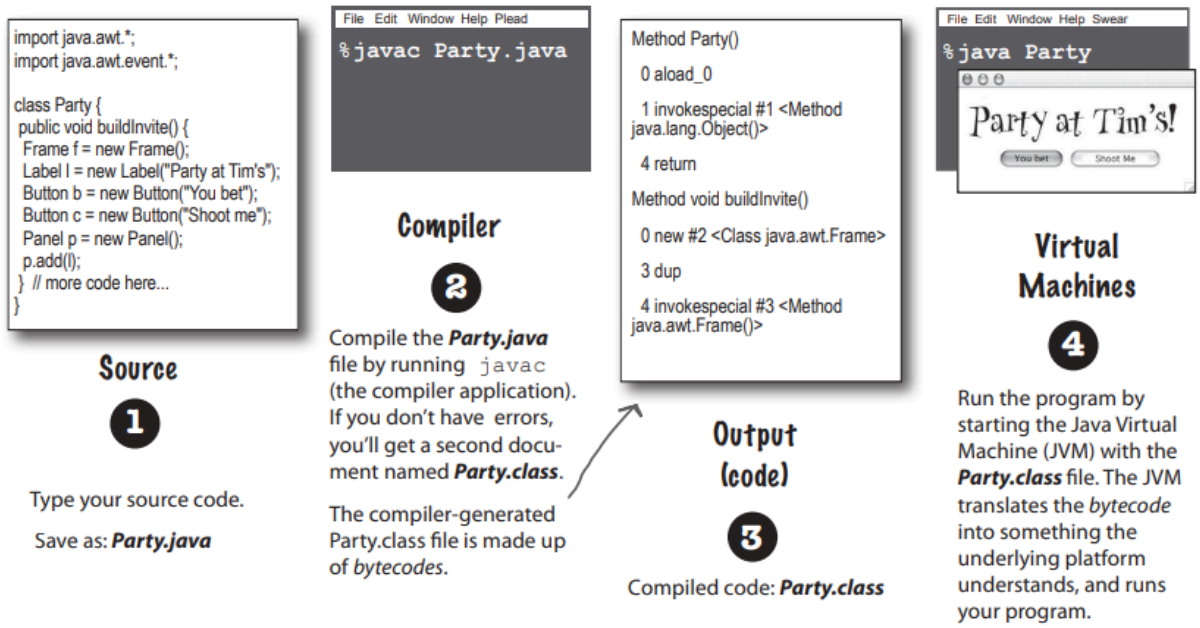
Head First Java Notes

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

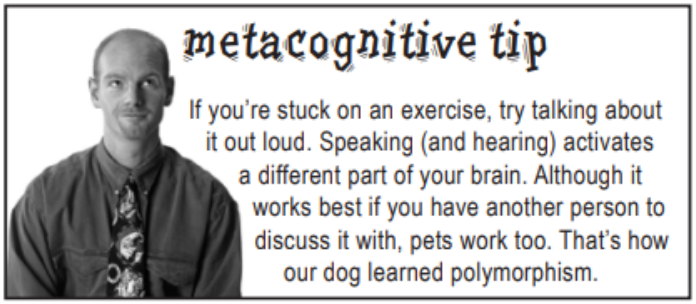
What do you like about OO?

"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."

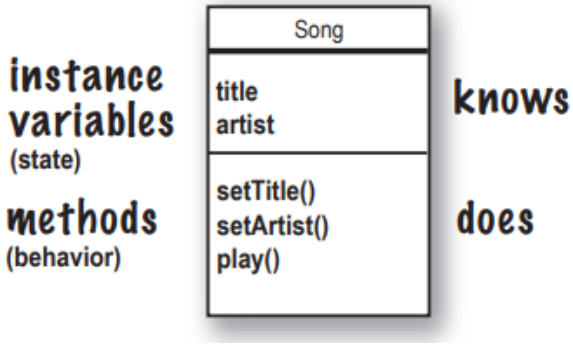
"Reusing code in other applications. When I write a new class, I can make it flexible enough to be used in something new, later."



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

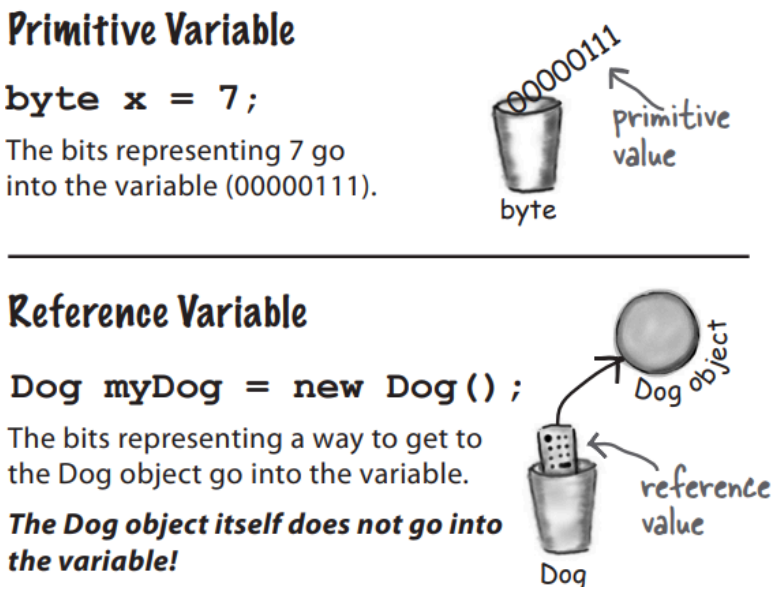
Type	Bit Depth	Value Range
boolean and char		
boolean	(JVM-specific)	***true*** or ***false***
char	16 bits	0 to 65535
numeric (all are signed)		
integer		
byte	8 bits	-128 to 127
short	16 bits	-32768 to 32767
int	32 bits	-2147483648 to 2147483647
long	64 bits	-huge to huge
floating point		
float	32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

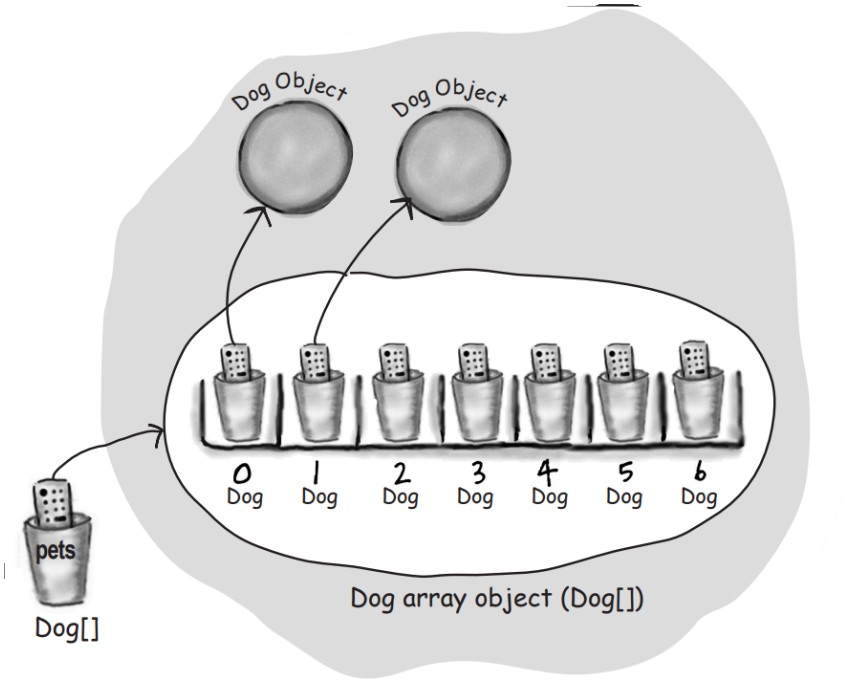
```
int x = 24;
```

```
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.



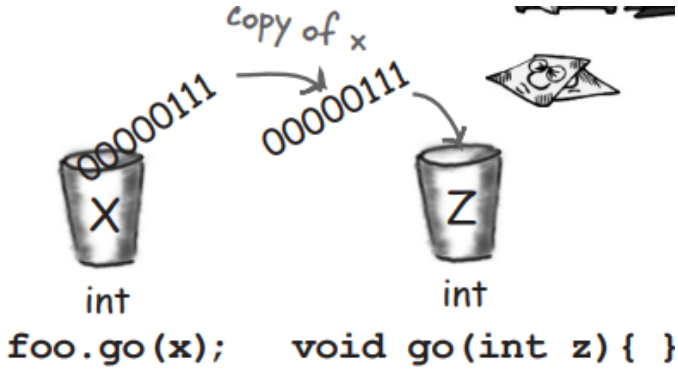
It also behaves same for the array that holds primitives

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.

Chapter 7 → Better Living in Objectville

Designing inheritance steps:

1. Look for objects that have common attributes and behaviors.
2. Design a class that represents the common state and behavior.
3. Decide if a subclass needs behaviors (method implementations) that are specific to that particular subclass type.
4. Look for more opportunities to use abstraction, by finding two or more subclasses that might need common behavior.
5. Finish the class hierarchy.

Using IS-A and HAS-A

When you want to know if one thing should extend another, apply the IS-A test.
Triangle IS-A Shape, yeah, that works.
Tub extends Bathroom, sounds reasonable. Until you apply the IS-A test.
To know if you've designed your types correctly, ask, "Does it make sense to say type X IS-A type Y?" If it doesn't, you know there's something wrong with the design, so if we apply the IS-A test, Tub IS-A Bathroom is definitely false.
What if we reverse it to Bathroom extends Tub?
Bathroom IS-A Tub doesn't work. Tub and Bathroom are related, but not through inheritance. Tub and Bathroom are joined by a HAS-A relationship.
Does it make sense to say "Bathroom HAS-A Tub"? If yes, then it means that Bathroom has a Tub instance variable. In other words, Bathroom has a reference to a Tub, but Bathroom does not extend Tub and vice versa.

```
// All these are seperate class files.

// Bathroom class
Tub bathtub;
Sink theSink;
```

```
// Tub class
int size;
Bubbles b;

// Bubbles class
int radius;
int colorAmt;

// Bathroom HAS-A Tub and Tub HAS-A Bubbles. But nobody inherits from (exter
```

The inheritance IS-A relationship works in only one direction!
Triangle IS-A Shape makes sense, so you can have Triangle extend Shape. But the reverse—Shape IS-A Triangle—does not make sense, so Shape should not extend Triangle.
Remember that the IS-A relationship implies that if X IS-A Y, then X can do anything a Y can do (and possibly more).

If class B extends class A, class B IS-A class A.
If class C extends class B, class C passes the IS-A test for both B and A.
Example:
Canine extends Animal, Wolf extends Canine then Canine IS-A Animal, Wolf IS-A Canine and Wolf IS-A Animal are true.

With polymorphism, the reference type can be a superclass of the actual object type

In other words, anything that extends the declared reference variable type can be assigned to the reference variable.
This lets you do things like make
polymorphic arrays.
Ex -

```
Animal[] animals = new Animal[5]; // Declare an array of type Animal
animals[0] = new Dog();  // you can put ANY subclass of Animal in the Animal arr
animals[1] = new Cat();
animals[2] = new Wolf();
```

```
animals[3] = new Hippo();
animals[4] = new Lion();
for (Animal animal : animals) {
 animal.eat();
 animal.roam();
}
```

You can have polymorphic arguments and return types.

```
// Polymorphic argument example
class Vet {
    public void giveShot(Animal a) { // the 'a' parameter can take any Animal type
        a.makeNoise();
    }
}

class PetOwner {
 public void start() {
    Vet vet = new Vet();
    Dog dog = new Dog();
    Hippo hippo = new Hippo();
    vet.giveShot(dog);
    vet.giveShot(hippo);
 }
}

// This is a polymorphic return type example because it returns a superclass Anir
class PetStore {
   public Animal getPet(boolean wantDog) {
       if (wantDog) {
           return new Dog();    // Dog IS-A Animal
       } else {
           return new Hippo();  // Hippo IS-A Animal
       }
```

```
    }
}
```

If I write my code using polymorphic arguments, where I declare the method parameter as a superclass type, I can pass in any subclass object at runtime. Cool. Because that also means I can write my code, go on vacation, and someone else can add new subclass types to the program and my methods will still work.

Keeping the contract: rules for overriding

Remember, the compiler looks at the reference type to decide whether you can call a particular method on that reference.
Appliance appliance = new Toaster(); (Appliance is reference type and Toaster is object type)
With an Appliance reference to a Toaster, the
compiler cares only if class Appliance has the method you're invoking on an Appliance reference. But at runtime, the JVM does not look at the reference type (Appliance) but at the actual Toaster object on the heap.
So if the compiler has already approved the method call, the only way it can work is if the overriding method has the same arguments and return types.

Basically reference type (compile time) defines what variables and methods are available to the reference variable and the object type (run time) determines which version of an overridden method is executed.

1. **Arguments must be the same, and return types must be compatible.**

The contract of superclass defines how other code can use a method. Whatever the superclass takes as an argument, the subclass overriding the method must use that same argument. And whatever the superclass declares as a return type, the overriding method must declare either the same type or a subclass type.

2. **The method can't be less accessible.**

That means the access level must be the same, or friendlier. You can't, for example, override a public method and make it private. What a shock that would be to the code invoking what it thinks (at compile time) is a public method, if suddenly at runtime the JVM slammed the door shut because the overriding version called at runtime is private!

Overloading a method

An overloaded method is just a different method that happens to have the same method name. It has nothing to do with inheritance and polymorphism. An overloaded method is NOT the same as an overridden method.

1. **The return types can be different.**
2. **You can't change ONLY the return type.**
3. **You can vary the access levels in any direction.**

Bullet Points:

▼ A subclass extends a superclass.

▼ Inherited methods can be overridden; instance variables cannot be overridden (although they can be redefined in the subclass, but that's not the same thing, and there's almost never a need to do it.)

▼ When a method is overridden in a subclass, and that method is invoked on an instance of the subclass, the overridden version of the method is called. (The lowest one wins.)

Chapter 8 → Serious Polymorphism

We know we can say:

```
Wolf aWolf = new Wolf();        // A Wolf reference to a Wolf object.
Animal aHippo = new Hippo();    // Animal reference to a Hippo object.

// But here's where it gets weird:
Animal animal = new Animal();   //Animal reference to an Animal object.
// what the heck does an Animal object look like? So make the class abstract.
```

The compiler won't let you instantiate an abstract class

An abstract class means that nobody can ever make a new instance of that class. You can still use that abstract class as a declared reference type, for the purpose

of polymorphism (to use it as a polymorphic argument or return type, or to make a polymorphic array).

When you're designing your class inheritance structure, you have to decide which classes are abstract and which are concrete. Concrete classes are those that are specific enough to be instantiated or it just means that it's OK to make objects of that type.

Abstract methods

Besides classes, you can mark methods abstract, too. An abstract class means the class must be extended; an abstract method means the method must be overridden.

Abstract methods don't have a body; they exist solely for polymorphism. That means the first concrete class in the inheritance tree must implement all abstract methods.
Ex- public abstract void eat();
If you declare an abstract method, you MUST mark the class abstract as well. You can't have an abstract method in a non-abstract class.

Why not make a class generic enough to take anything?

Every class in Java extends class Object. Any class that doesn't explicitly extend another
class, implicitly extends Object. Simply we can say every class in Java is either a direct or indirect subclass of class Object (java.lang.Object).

The Object class serves two main purposes: to act as a polymorphic type for methods that need to work on any class that you or anyone else makes, and to provide real method code that all objects in Java need at runtime (and putting them in class Object means all other classes inherit them) like toString(), hashcode(), getClass(), equals(Object o) and others.

Q. If it's so good to use polymorphic types, why don't you just make ALL your methods take and return type Object?

For one thing, you would defeat the whole point of "type-safety," one of Java's greatest protection mechanisms for your code. With type-safety, Java guarantees that you won't ask the wrong object to do something you meant to ask of another

object type. Like, ask a Ferrari (which you think is a Toaster) to cook itself.
But the truth is, you don't have to worry about that fiery Ferrari scenario, even if you do use Object references for everything. Because when objects are referred to by an Object reference type, Java thinks it's referring to an instance of type Object. And that means the only methods you're allowed to call on that object are the ones declared in class Object! So if you were to say:

```
Object o = new Ferrari();
o.goFast(); //Not legal!
// You wouldn't even make it past the compiler.
```

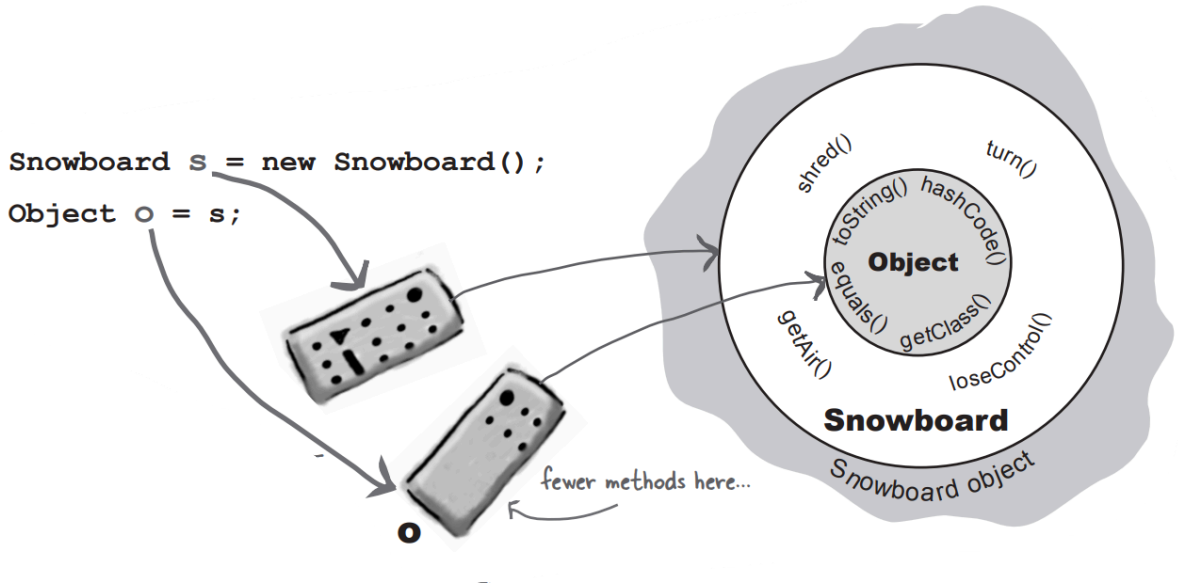
Because Java is a strongly typed language, the compiler checks to make sure that you're calling a method on an object that's actually capable of responding.
In other words, you can call a method on an object reference only if the class of the reference type actually has the method.

Using polymorphic references of type Object has a price...

Everything comes out of an ArrayList<Object> as a reference of type Object, regardless of what the actual object is or what the reference type was when you added the object to the list. A cast can be used to assign a reference variable of one type to a reference variable of a subtype.
We can cast the Object, so we can treat it like he really is and call it's method -

```
ArrayList<Object> myDogArrayList = new ArrayList<Object>(); // Object is type p
Dog d = new Dog();
myDogArrayList.add(d);

Object o = myDogArrayList.get(0);
// If you're not sure about object returned is an instance of Dog or Cat use this
if(o instanceOf Dog) {
   Dog dog = (Dog) o;
}
```



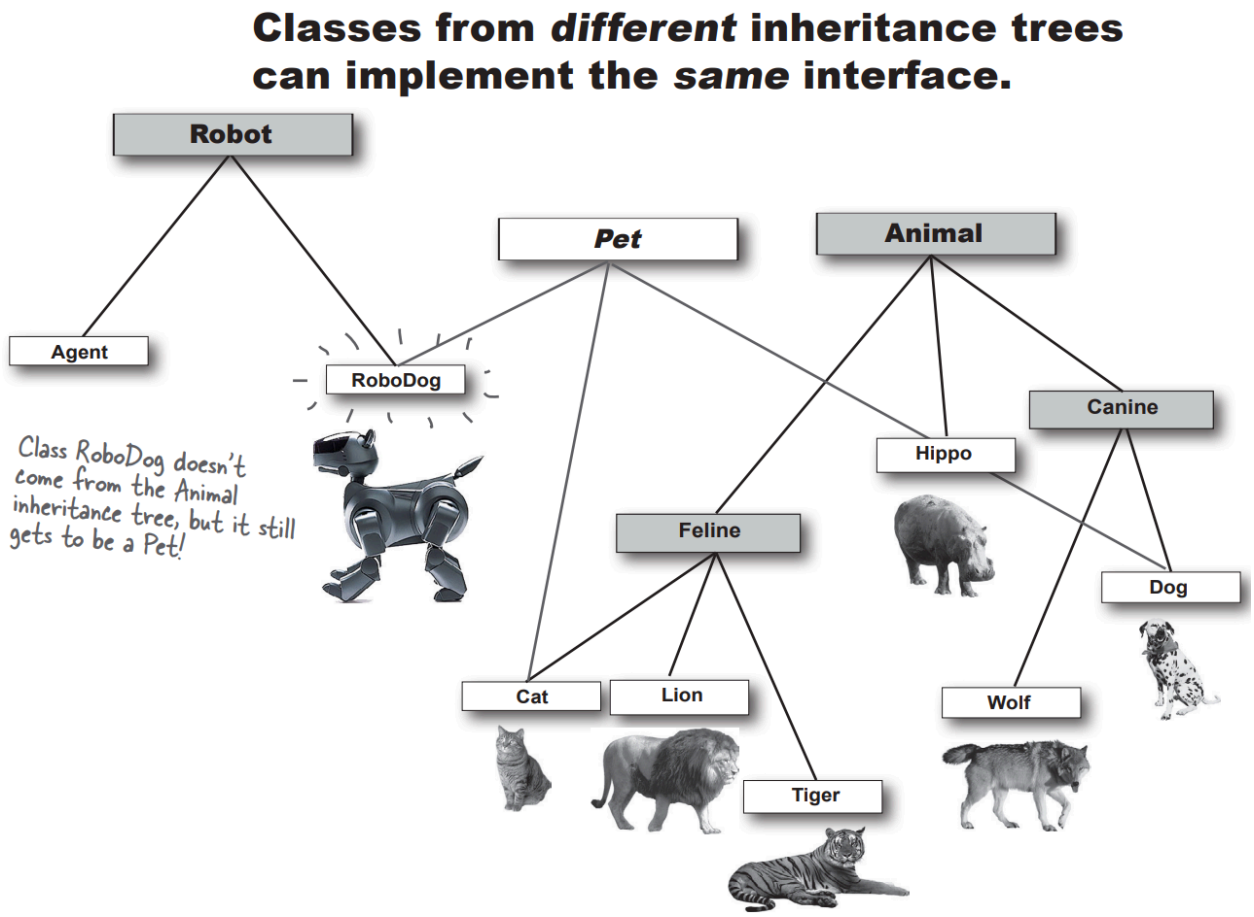
The compiler decides whether you can call a method based on the reference type, not the actual object type.

Interface

A Java interface solves your multiple inheritance problem. A Java class can have only one parent (superclass), and that parent class defines who you are. But you can implement multiple interfaces, and those interfaces define roles you can play.

```
public interface Pet {
 //Interface methods are implicitly public and abstract, so typing it is optional.
 public abstract void beFriendly();
 public abstract void play();
}

//  a class can implement multiple interfaces!
public class Dog extends Animal implements Pet, Saveable, Paintable { ... }
```



Dark colored classes are abstract classes while Pet is an interface. Other are concrete classes.

The main purpose of interfaces is polymorphism, polymorphism, polymorphism. When you use a class as a polymorphic type (like an array of type Animal or a method that takes a Canine argument), the objects you can stick in that type must be from the same inheritance tree. But not just anywhere in the inheritance tree; the objects must be from a class that is a subclass of the polymorphic type. An argument of type Canine can accept a Wolf and a Dog, but not a Cat or a Hippo. But when you use an interface as a polymorphic type (like an array of Pets), the objects can be from anywhere in the inheritance tree. The only requirement is that the objects are from a class that implements the interface. Allowing classes in different inheritance trees to implement a common interface is crucial in the Java API.

How do you know whether to make a class, a subclass, an abstract class, or an interface?

▼ Make a class that doesn't extend anything (other than Object) when your new class doesn't pass the IS-A test for any other type.

▼ Make a subclass (in other words, extend a class) only when you need to make a more specific version of a class and need to override or add new behaviors.

▼ Use an abstract class when you want to define a template for a group of subclasses, and you have at least some implementation code that all subclasses could use. Make the class abstract when you want to guarantee that nobody can make objects of that type.

▼ Use an interface when you want to define a role that other classes can play, regardless of where those classes are in the inheritance tree.

```
class BuzzwordsReport extends Report {
 void runReport() {
    super.runReport(); // superclass version of method imports stuff that subclass
    printReport();
 }
}
```

Sometimes the concrete code isn't enough to handle all of the subclass-specific work. So the subclass overrides the method and extends it by adding the rest of the code. The super keyword is really a reference to the superclass portion of an object. When subclass code uses super, as in super.runReport(), the superclass version of the method will run.

Chapter 9 → Life and Death of an Object

Bullet Points:

▼ Java has two areas of memory we care about: the Stack and the Heap.

▼ All local variables live on the stack, in the frame corresponding to the method where the variables are declared.

▼ Object reference variables work just like primitive variables—if the reference is declared as a
local variable, it goes on the stack.

▼ All objects live in the heap, regardless of whether the reference is a local or instance variable.

▼ Instance variables live within the object they belong to, on the Heap.

▼ If the instance variable is a reference to an object, both the reference and the object it refers to are on the Heap.

▼ A constructor is the code that runs when you say new on a class type. A constructor must have the same name as the class, and must not have a return type.

▼ If you don't put a constructor in your class, the compiler will put in a default constructor which is always a no-arg constructor.

▼ If you put a constructor—any constructor—in your class, the compiler will not build the default constructor.

▼ All the constructors in an object's inheritance tree must run when you make a new object.

▼ Remember, a subclass might inherit methods that depend on superclass state (instance variables). For an object to be fully formed, all the superclass parts of itself must be fully formed, and that's why the superclass constructor must run first.

▼ When a constructor runs, it immediately calls its superclass constructor, all the way up the chain until you get to the class Object constructor.

▼ A new Hippo object also IS-A Animal and IS-A Object. If you want to make a Hippo, you must also make the Animal and Object parts of the Hippo. This all happens in a process called Constructor Chaining.

▼ If you don't provide a constructor, then the compiler puts one that looks like -

```
public ClassName() {
        super();
}
```

▼ If you do provide a constructor but you do not put in the call to super() then the compiler will put a call to super() in each of your overloaded constructors. The compiler inserted call to super() is always a no-arg call.

▼ The call to super() must be the first statement in each constructor! Because the superclass parts of an object have to be fully formed (completely built) before the subclass parts can be constructed.

▼ Use this() to call a constructor from another overloaded constructor in the same class.

▼ The call to this() can be used only in a constructor, and must be the first statement in a constructor.

▼ A constructor can have a call to super() OR this(), but never both!

▼ An object's life depends entirely on the life of references referring to it. An object becomes eligible for GC when its last live reference disappears.

▼ Three ways to get rid of an object's reference:

1. The reference goes out of scope, permanently

```
void go() {
        Life z = new Life();   // reference 'z' dies at end of method.
}
```

2. The reference is assigned another object

```
Life z = new Life();
z = new Life();        // the first object is abandoned when z is
```

'reprogrammed' to a new object.

3. The reference is explicitly set to null

```
Life z = new Life();
z = null;              // the first object is abandoned when z is 'deprogrammed'.
```