Chapter 2

We have some idea of what a program is and how we go about the activity of programming.

Let's take a step back and look at the big picture.

How do we write programs to solve problems and accomplish tasks?

Java is an "object-oriented" language.

(This means that we define programs in terms of "objects" and their interactions.)
Idea: identify which "objects" are involved in a problem domain and how they might interact.

Example: if your problem is "I need to manage a bank," the objects involved are accounts, customers, currency, etc.

Then, write a program by defining the objects, defining their interactions, and letting the objects do the work.

Objects correspond to problem-domain entities and interact with one another.

A class is a template for a kind of object.

(We say that objects are instances of classes. You can also think of classes as concepts.)

Examples

<table>
<thead>
<tr>
<th>Class (or concept)</th>
<th>Object (or instance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>checking account</td>
<td>my checking account</td>
</tr>
<tr>
<td>positive integer</td>
<td>42</td>
</tr>
<tr>
<td>building</td>
<td>the computer science building</td>
</tr>
<tr>
<td>rectangle</td>
<td></td>
</tr>
</tbody>
</table>

Objects are collections of data and sets of operations.

Accessor methods: inspect the data of an object.

Mutator methods: modify the data of an object.
Your programs will create and manipulate objects to model problems.

Let's look at our simple program in greater detail.

Classes, methods, and statements comprise the anatomy of a program.

```java
public class Hello {
    public static void main(String[] args) {
        String greeting = "Hello, world!";
        String yellingGreeting = greeting.toUpperCase();
        System.out.println(greeting);
        System.out.println(yellingGreeting);
    }
}
```

Classes are "units" of code containing methods. Methods are sequences of simple instructions. Statements are like sentences: single, imperative instructions.

Let's talk about statements a bit.

Statements are made up of identifiers and operators.
Identifiers provide names for "things in your program"

Identifiers are case-sensitive, must follow rules and should follow conventions

Identifier rules make it possible for the Java compiler to understand your program.

Naming conventions make it possible for future programmers to understand your program!

Rule #1 of maintenance programming: at some point, you will be the "future programmer" for some code you wrote. Make it easy on yourself.

Identifiers must follow rules:

- Can be made up strictly of letters, digits, and the underscore character (\_). May not contain spaces or other characters.
- The first letter must not be a digit.
- Identifiers are case-sensitive.
- That is, \_Fred is not the same as Fred.
- Finally, reserved words may not be used as identifiers.
Reserved words

- Some names have been reserved by the Java language
- Examples: public, static, void, class
- Eclipse will highlight these by making them purple
- You can't use these names for identifiers

Examples

- Legal
  - fred, harry, Fred, fido, Fred123, Fred123
  - Person, Person2

- Not legal
  - class, 2legit2quit, bill1, dot.com, $10

Identifiers provide names for "things in your program," like variables, classes, and methods.
(We've talked about classes a bit already, and we'll talk about methods soon.)

Naming conventions give other programmers a good idea of what sort of thing an identifier is naming!

Naming conventions

- Classes: nouns. The first letter of each word is capitalized.
  - examples: String, PrintStream
- Variables: nouns. The first letter of the first word is lowercase; the first letter of subsequent words is capitalized.
  - examples: greeting, highScore
- Methods: verbs, capitalized as variables.
  - examples: reset(), crashMyComputer()

Classes, methods, and statements comprise the anatomy of a program

```java
public class Hello {
    public static void main(String[] args) {
        String greeting = "Hello, world!";
        String yellingGreeting = greeting.toUpperCase();
        System.out.println(greeting);
        System.out.println(yellingGreeting);
    }
}
```
Statements

- We know that a statement consists of identifiers, operators and literals.
- We also know that a statement is terminated by a semicolon, just like a sentence in English ends with a period, exclamation point, or question mark.
- But what do some of the statements we’ve seen actually do?

This statement sends a message to an object called System.out

```
System.out.println(greeting);
```

(Objects can communicate with other objects by sending messages.)

This statement declares a variable, or a named place to store a value

```
String greeting = "Hello, world!";
```

(We pronounce = "gets")

Declaring variables

- Declaration: type id;
  - String foo;
  - int x;
- Declaration with definition: type id = val;
  - String foo = "foo";
  - int x = 42;
- Clicker y = new Clicker();

Variables are very important, so we’ll talk about them now.
A variable is a named place to store a value.

Every variable has a type.

Types define the range of values that a variable may hold.

Types also define the sorts of ways a value may be used.

Why are types a good idea?

The Java language supports two kinds of types.
A variable with a primitive type will hold a value of that type.

A variable with a reference type contains the location of an object.

How can we get the location of an object? Glad you asked!

Object references

- You can copy a reference to a preexisting object.
- You can get a reference to a newly-allocated object, too.

```java
String s = "Hello!";
String t = s;
```

String s and t refer to the same String.

Operator new creates a new object and returns a reference to it.
String s = "Hello!";
String t = new String(s);

1 and s refer to different Strings with the same contents.

Kinds of types

- Primitive types
  - byte, short, int, long
  - float, double
  - char
  - boolean
- Reference types
  - all object types, denoted by class names (e.g., String)

Remember that reference types do not hold objects -- they merely hold references to objects!

String s = "Hello, world!";
String t = s;
t = "Hello, friends!";

What happens now?

Clicker c = new Clicker();
Clicker d = c;
d.click();
System.out.println(c.getClicks());

What will this print?
We already know a bit about variables.

- A variable is a named place to store a value.
- A variable declaration consists of a type and an identifier.
  
  * e.g. `String fred;`

- It may also include an initial value.
  
  * e.g. `String fred = "fred";`

Remember that every Java statement ends with a semicolon!

We have a few questions left to answer about variables.

- What is the lifetime of a value?
- Where can I use a variable?
  
  Computer scientists call this the variable’s scope, by the way.

- What happens if a variable has not received a value before we try and use it?

The scope of a variable in Java is given by the enclosing code block (sequence of statements between matched curly braces).

```java
public class Hello {
    public static void main(String[] args) {
        String greeting = "Hello, World!";
        System.out.println(greeting);
    }
}
```

Block statements

```java
public class Hello2 {
    public static void main(String[] args) {
        String greeting = "Howdy!";
        {
            String greeting = "Hello!";
            System.out.println(greeting);
        }
    }
}
```

What about this...?

```java
public class Hello3 {
    public static void main(String[] args) {
        String greeting = "Howdy!";
        {
            String greeting = "Hello!";
        }
        System.out.println(greeting);
    }
}
```

...or this?

```java
public class Hello4 {
    public static void main(String[] args) {
        String greeting = "Howdy!";
        {
            String greeting = "Hello!";
        }
    }
}
```
What about variables that have been declared but not defined?

```java
public class Hello {
    public static void main(String[] args) {
        String greeting;
        System.out.println(greeting);
    }
}
```

Let’s talk about choosing objects to solve a problem.

Usually, example programs involve mundane business problems.

What objects are involved in a life-and-death problem?

What objects are involved? What data and methods should each object include?

What does the player need to know?

• score
• "lives"
• position
• how to rotate
• how to move
What does the foe need to know?

- posture
- whether to
- chase, flee, or
- return
- how to react

OK, we've got a better idea of how to think about objects. How can we use objects?

Objects

- Recall that an object is a collection of fields (viz., data) and methods that access and operate on that data
- We've already seen some objects in class:
  - System.out
  - greeting
- How did we use them?

Knowing about objects

- What do we know about System.out?
  - Not much!
  - We know how we can interact with it, but not how it does its work.
- Is this a big deal? (What do we know about a DVD player, or an ATM?)

Using objects

- You interact with objects by sending messages to them.
  - We also call this invoking methods on them.
  - To do this, use the dot notation, e.g.
    - System.out.println(greeting);
  - greeting.toUpperCase();
- We call this message sending or method invocation.

Different kinds of objects

- Note that greeting supports different methods than System.out
- It's not hard to imagine that they have different collections of data as well
- The methods that can be invoked on an object are defined by its class.
Remember that a class defines a template for a kind of object.

A simple class

```java
public class Clicker {
    private int clicks;

    public void click() {
        clicks += 1;
    }

    public int inspect() {
        return clicks;
    }

    public void reset() {
        clicks = 0;
    }
}
```

Public interface

- Programmers who use Clicker objects, though, won't see what you just saw.
- Rather, they will simply see the public interface.
- A public interface specifies what you can do with instances of a class -- not how those tasks are implemented!

A method is a sequence of instructions that operates on the data of an object

- Recall that a method is a sequence of instructions that operates on the data of an object.
- Methods can also take parameters and return values.

Parameters are inputs to a method.

- Why might you want to provide inputs to a method?
- Every method of an object may or may not take explicit parameters; those that are named in the method signature.
- NB: we refer to the values supplied to a method call as the arguments.

Return values

- A method may return a value as well.
- Type specified in method signature.
- Value specified via the return reserved word.
- What if a method returns no value?
public int inspect() {
    return clicks;
}

Accessor methods inspect an object's state.

Mutator methods modify an object's state.

Creating objects

- Remember that we create objects with operator `new`
  - Clicker c = new Clicker();
  - `new` does three things:
    - Allocates space for a new object of a certain type
    - Initializes the object's fields to default values
    - Invokes the object's constructor

What's a constructor?

Constructors

- Constructors are "special" methods:
  - A constructor has no return value (not even void)
  - You can't create a "ctor" directly; Java does it for you
  - You can use the `new` operator
  - A constructor initializes the fields (state) of a new instance

Example ctors

```java
public class Clicker {
    private int clicks;
    public Clicker() {
        clicks = 0;
    }
    public Clicker(int clicks) {
        this.clicks = clicks;
    }
    // ...other methods
}
```
Just to make sure you’re paying attention: What happens when you create an object?

Dogotto = new Dog("Otto", "otto.jpg");

Creating an object
Dogotto = new Dog("Otto", "otto.jpg");

1. The Java VM sets aside a Dog-shaped chunk of RAM.
2. The JVM initializes the fields of the new Dog to the default values for each type.
3. The JVM executes user constructor code to fill in the values for each field. (This may include constructing additional objects!)
4. Finally, the JVM gives otto a reference to the newly-allocated Dog.

Any questions?

We know how to create an object.

Notice there’s no way to delete an object.

Garbage collection is the process used by the Java Virtual Machine to reclaim the memory used by unneeded objects.
Remember that local variables exist only for the duration of a method invocation.

Sidebar: methods

- When you invoke a method, Java automatically creates space for its local variables, including this and parameter values.
- When that method returns, that space goes away as the variables go out of scope (think of taking the top plate off a stack)
- The next slide shows how it works (this and parameters are omitted for clarity)

Sidebar: methods

```java
public int fred(int i) {
    int x = 5 * i;
    int y = 4 * i;
    return barney(x + y);
}
```

```java
public int barney(int i) {
    int x = i % 8;
    return x;
}
```

```
int z = fred(4);
```

```
public int fred(int i) {
    int x = 5 * i;
    int y = 4 * i;
    return barney(x + y);
}
```

```
public int barney(int i) {
    int x = i % 8;
    return x;
}
```

Some review questions

- What is an object?
- What does it mean for Java to be object-oriented?
- What is a class?
Review: identifiers
- What is an identifier?
- What are identifiers used for?
- What are the rules governing identifier names?
- What are the conventions that Java programmers follow in choosing names?

Review: variables
- What is a variable?
- How do we declare a variable?

Review: types
- What is a type?
- Why is it a good idea for variables to have types?