Execution flows from one method to another

- The called method can be within the same class, in which case only the method name is needed.
When designing methods, think about their use

- Public methods describe the services that this class provides to other classes
  - Public methods are often called **service methods**

- Methods that simply return information about an object are called **accessor methods** (they allow other objects to access information about this object’s state)

- Methods that modify an object are called **mutator methods** (they allow other objects to change or mutate this one)

- Private methods are often called **support methods** or **helper methods**, because they help perform the work provided by service methods
Control also flows between classes and back

- The called method can be part of another class or object
Decompose larger methods into cohesive pieces

- A method should be **relatively small**, so that it can be understood as a single entity
  - No longer than 40 lines
  - A better goal: fits completely on the screen at once (about 20 lines or less)
- A potentially large method should be decomposed into several smaller methods as needed for clarity
- A service method of an object may call one or more support methods to accomplish its goal
- Support methods could call other support methods if appropriate
Abstraction and modularization are critical

- Whether designing a collection of classes that work together, or a collection of methods within a class, these concepts are central:

  - **Abstraction** is the ability to ignore details of parts to focus attention on a higher level of a problem.

  - **Modularization** is the process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways.
Primitive types differ from object types

SomeObject obj;

int i;

object type

primitive type

32
Assignment differs for primitive and object types

```java
SomeObject a;

int a;
32

SomeObject b;

int b;
32

b = a;
```
Here’s Binky to explain it a different way

Pointer Fun with Binky

by Nick Parlante
This is document 104 in the Stanford CS Education Library — please see cslibrary.stanford.edu for this video, its associated documents, and other free educational materials.

Copyright © 1999 Nick Parlante. See copyright panel for redistribution terms.
Carpe Post Meridiem!
Object identity is not the same as equality

Consider two distinct objects:

```
:Person
  "Fred"
```

```
:Person
  "Jill"
```

```
person1

person2
```

```
person1 == person2?
```
Objects with the same value aren’t the same object

Two distinct objects with the same field values:

```
:Person
  "Fred"
person1

:Person
  "Fred"
person2
```

```
person1 == person2 ?
```
Two variables can refer to the same object

These variables refer to the identical object:

```
:Person
 "Fred"
```

```
:Person
 "Fred"
```

```
person1
```

```
person2
```

```
person1 == person2 ?
```
Identity vs equality is important for Strings

String input = reader.getInput();
if ( input == "bye" )
{
    ...
}

== tests identity, not equality

>>>> (may be) false!
String input = reader.getInput();
if ( input.equals( "bye" ) ) {
  ...
}

equals() tests equality

equals "bye"

input

true!
You can use equals() on any pair of objects

- equals() is a message that all objects understand
- It is always preferable to ==, except when you want to check whether two variables refer to the same object
- But **beware**!
  - Each class must define its own notion of equals()
  - Most library classes do this, but ...
  - If a class does not define what equals() means, it defaults to be the same thing as ==
  - Many `java.util` classes define equals() to mean “shallow” equality (i.e., two objects are equal if their fields are == to each other)