Introduction to Interactive Programming
by Lynn Andrea Stein
A Rethinking CS101 Project

# Building New Things: Classes and Objects

## Chapter Overview

- How do I group together related rules?
- How do I build a computational object?
- What are Java programs ***really*** made of?

In this chapter, you will learn to put together the pieces you've already seen -- things, names, expressions, statements, rules, and interfaces -- to create computational objects that can populate your communities.

In order to create an individual object, you first have to describe what kind of object it is. This includes specifying what things you can do with it -- as in its interface(s) -- but also how it will actually work. This description of the "kind of object" is like building a recipe for the object, but not like the object itself. (You can't eat the recipe for chocolate chip cookies.) These object-recipes are called classes.

For each thing that your object can do, your class needs to give a rule-recipe. This is called a method. Your objects may also have (named) pieces. These are called fields, and they are special java names that are always a part of any object made from this recipe.

When you actually use your class (recipe) to create a new object, there may be things that you need to do to get it started off right. These startup instructions are called a constructor.

When you are building an object, you are bound by the interfaces it promises to meet. If the interface promises a behavior, you have to provide a rule (method) body for the object to use.

This chapter is supplemented by reference charts on the syntax and semantics of Java classes, methods, and fields. It includes style sidebars on good documentation practice.

Most of the syntax of this section is covered in the appendix Java Charts.

## Objectives of this Chapter

1. To recognize the difference between classes and their instances.
2. To be able to read a class definition and project the behavior of its instances.
3. To be able to define a class, including its fields, methods, and constructors.

## Classes are Object Factories

In a previous chapter, we saw how to build an interface, or specification, that described the contract a particular kind of object would fulfill. We also saw that an interface does not provide enough information to actually create an object of the appropriate kind. Interfaces do not say anything about how methods actually work. They do not talk about the information that an object needs to keep track of. And they do not say anything about the special things that need to happen when a new object is created.

In this chapter, we will learn how to create objects and how to describe the ways in which they work. The mechanism that Java provides for doing this is called a **class**. Like an interface, a class says something about what kind of thing an object is. Like an interface, a class defines a Java type. However, interfaces specify only contracts; classes also specify implementation. Class methods are full-fledged rules, with bodies telling how to accomplish the task of that rule (not just the rule specification, or method signature, of abstract interface methods). Classes also talk about data -- information to be kept track of by objects -- as well as methods, or behavior. And a special part of a class -- the constructor -- talks about how to go about creating an object of the type specified by that class.

## Classes and Instances

Objects created from a class are called **instances** of that class. For example, the class `CheckBox` refers to the instructions for creating and manipulating a GUI widget that displays a selectable checkbox on your computer screen. `CheckBox` is the name of the class, i.e., of the instructions. Let's say we create two particular checkboxes:

```
CheckBox yesCheckBox = new CheckBox();
CheckBox  noCheckBox = new CheckBox();
```

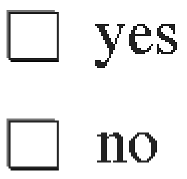

Figure 1. The actual `CheckBox`es.

The two objects labeled by the names `yesCheckBox` and `noCheckBox` are instances of the class `CheckBox`. That is, they are *particular* `CheckBox`es. The instructions for how to create -- or be -- a `CheckBox`, on the other hand, aren't a `CheckBox` at all; the instructions are *instructions*, or a class. In fact, the instructions are an object, too, though a very different kind of object and not one as obviously useful as a `CheckBox` or a `Timer` or a `Counter`. The kind of object the instructions are is called a `Class`.

Because the class contains the instructions for how to make a new instance and for how to behave like an instance of that class, we sometimes say that a class is like a **factory** where instances are made. Both a factory and its product are objects, but factories and the widgets that they make are very different kinds of objects. The factory has all of the know-how about its instances. But the factory isn't one of its instances, just as the class `CheckBox` isn't a `CheckBox`. It's a factory!

### Recipes Don't Taste Good

Another analogy for a class (as opposed to its instances) is that the class is like a **recipe** for how to make instances. The instances are like food cooked from the recipe (say, chocolate chip cookies). It isn't hard to tell the difference between these things. The cookies smell good. If you are hungry, the note-card with the recipe on it won't be very satisfying. (It probably tastes a lot like cardboard.) On the other hand, if you're going over to Grandma's to cook, you might want to take the recipe but you probably don't want to stick the chocolate chip cookie in your back pocket. Classes actually contain a lot of information other than just how to make an instance. (The recipe might, too. It might include information on how long it takes to make the cookies, whether they need to be refrigerated, how long it will take before they go stale, or even how many calories they contain.)

Figure 2. Two recipes (classes) and two platefuls of cookies (instances) made from the second recipe.

### Classes are Types

Like interfaces, classes represent particular *kinds* of objects, or types. Once a class has been defined (see below), its name can be used to declare variables that hold objects of that type. So an instance of a class can be labeled using a name whose declared type is that class. For example, the `CheckBox`es described above are labeled using names (`yesCheckBox` and `noCheckBox`) whose declared type is `CheckBox`. Note that the class `CheckBox` -- the `CheckBox` recipe -- can't be labeled using a name whose declared type is `CheckBox`. The type of the class `CheckBox` is `Class`, not `CheckBox`. (This is the recipe vs. cookie distinction again.)

If an object is an instance of a class -- such as `yesCheckBox` and the class `CheckBox` -- then the type membership expression (`yesCheckBox instanceof CheckBox`) has the value `true`. Of course, `CheckBox instanceof CheckBox` is `false` (since the class isn't a `CheckBox`), but `CheckBox instanceof Class` is `true`.

**Style Sidebar**

## Class Declaration

It is conventional to declare the members of a class in the following order:

- static final fields (i.e., constants)
- static non-final fields

- non-static fields
- constructors
- methods

This order is not necessary -- any class member can refer to any other class member, even if it is declared later -- but it makes your code easier to read and understand.

All non-private members of the class should be listed in the class's documentation.

## Class Declaration

A class definition starts out looking just like an interface declaration, although it says that it is a class rather than an interface:

```
class Cat {
    ....
}
```

A class definition tells you what type of thing it is -- a class -- what it is called -- its name -- and what it's made of -- its definition, between braces. This last part is called the class's **body**. The body of the class definition contains all of the information about how instances of that class behave. It also gives instructions on how to create instances of the class. These elements -- fields, methods, and constructors -- are called the class's **members**. [Footnote: Be careful not to confuse members, which are parts of the class, with instances, which are objects made from the class. If chocolate chip cookies are instances of the cookie class (recipe), the chocolate chips are members of the class.] Each member is declared inside the body of the class, but not inside any other structure within the class. Another way of saying this is that each member is declared at **top level** within the class. So members are all and only those things declared at top level within a class.

For example, each instance of Java's `Rectangle` class has a set of four coordinates describing the rectangle's position and extent, as well as methods including one which tells whether a particular x, y pair is `inside` the `Rectangle`.

```
...class Rectangle {

     ...
     int height;
     int width;
     int x;
     int y;
     ...
     ...inside(...)...

}
```

In this case, `height`, `width`, `x`, `y`, and `inside` are all members of the `Rectangle` class.

Members and instances are quite different:

- members are parts of a class
- instances are things created from the class.

We will return to each of the elements of this declaration later in this chapter.

**Classes and Interfaces**

A class may **implement** one or more interfaces. This means that the class subscribes to the promises made by those interfaces. Since an interface promises certain methods, a class implementing that interface will need to provide the methods specified by the interface. The methods of an interface are abstract -- they have no bodies. Generally, a class implementing an interface will not only match the method specifications of the interface, it will also provide bodies -- implementations -- for its methods.

For example, a `ScoreCounter` class might meet the contract specified by the `Counting` interface:

```
interface Counting
{
    abstract void increment();
    abstract int getValue();
}
```

So might a `Stopwatch`, although it might have a totally different internal representation. Both would have `increment()` and `getValue()` methods, but the bodies of these methods might look quite different. For example, a ScoreCounter for a basketball game might implement increment() so that it counts by 2 points each time, while a Stopwatch might call its own increment() method even if no one else does.

A class that implements a particular interface must declare this explicitly:

```
class ScoreCounter implements Counting {
    ....
}
```

If a class implements an interface, an instance of that class can also be treated as though its type were that interface. For example, it can be labeled with a name whose declared type is that interface. For example, an instance of class `ScoreCounter` can be labeled with a name of type `Counting`. It will also answer true when asked whether it's an `instanceof` that interface type: if `myScoreCounter` is a `ScoreCounter`, then `myScoreCounter instanceof Counting` is true. Similarly, you can pass or return a `ScoreCounter` whenever a `Counting` is required by a method signature.

The generality of interfaces and the inclusion of multiple implementations within a single (interface) type is an extremely powerful feature. For example, you can use a name of type `Counting` to label either an instance of `ScoreCOunter` or an instance of `Stopwatch` (and use its `increment()` and `getValue()` methods) without even knowing which one you've got. This is the power of interfaces!

## Data Members, or Fields

The `Rectangle` class, above, had certain things that were a part of each of its instances: `width`, `height`, etc. This is because part of what it is to be a `Rectangle` involves having these properties. A

`Rectangle`-factory (or `Rectangle`-recipe) needs to include these things. Of course, each `Rectangle` made from this class will have its *own* width, height, etc. -- it wouldn't do for every `Rectangle` to have the *same* width!

[Insert pic of rectangles]

Many objects have properties such as these: information called **state** or **data** that each instance of a class needs to keep track of. This kind of information is stored in parts of the object called fields. A **field** is simply a name that is a part of an object. For the most common kind of field, each instance of a class is born with its own copy of the field -- its own label or shoebox, depending on the type of name the field is.

Declaring a field looks just like an ordinary name declaration or definition (depending on whether the field is explicitly initialized). Such a declaration is a field declaration if it takes place at **top level** in the class, i.e., if it is a class member. (A local variable declared inside a method body or other block is not at top level in the class.)

Consider the `Rectangle` class defined above and reproduced here:

```
class Rectangle {

     int height;
     int width;
     int x;
     int y;
     ...

}
```

Each instance of this class will have four `int`-sized shoeboxes associated with it, corresponding to the height, width, horizontal and vertical coordinates of the `Rectangle` instance. These fields are declared at top level inside the class body.

These fields are declared here, but not initialized: none of these fields is explicitly assigned a value. Fields, unlike variables, are initialized by default. If you don't give a field a value explicitly, it will have a default value determined by its type. For example, `int` fields have a default value of `0`. Contrast `int` local variables, which don't have a default value and cannot be used until they are initialized. For details on the default values for each type, see the sidebar on Default Initialization.

## Java Types and Default Initialization

In Java, field names can be declared without assigning them an initial value. In this case, Java automatically provides the field with a **default value**. The value used by Java depends on the type of the field.

Fields with numeric types are initialized by default to the appropriate `0`; that is, either `0` or `0.0` (using the appropriate number of bits).

Fields with type `char` default to the value of the character with ascii and unicode code 0 -- `'\u000'`. This character is sometimes called **the null character**, but should not be confused with the special Java value `null`, the non-pointer.

Fields with `boolean` type are by default assigned the value `false`.

Fields associated with reference types -- including `String` -- are by default not bound to any object, i.e., their default value is `null`.

If a declaration is combined with an assignment -- i.e., a definition -- the definition value is used and these default rules do not apply.

These rules apply to names of fields as well as to the components of arrays -- described in a later chapter. In contrast, local variables must be explicitly assigned values -- either in their declaration (definition) or in a subsequent assignment statement -- before they are used. There are also names called parameters, which appear in methods and `catch` expressions; they are initialized by their invoking expressions and are discussed in elsewhere in this book.

## Fields are not Variables

The difference in default initialization is only one difference between fields and local variables. This section covers several other important differences after first reviewing some properties of local variables.

A local variable is a name declared inside a method body. The scope of a local variable -- the space within which its name has meaning -- is only the enclosing block. At most, this is the enclosing method, so the maximum lifetime of a variable name is as long as the method is running. Once the method exits, the variable goes away. (A similar variable will come into existence the next time the method is invoked, but any information stored in the variable during the previous method invocation is lost.)

### Hotel Rooms and Storage Rental

Because a field is a part of an object, and because an object continues to exist even when you're not explicitly manipulating it, fields provide longer-term (persistent) storage. When you exit a block, any variables declared within that block are cleared away. If you reenter that block at some later point, when you execute the declaration statement, you will get a brand new variable. This is something like visiting a hotel room. If I visit Austin frequently, I may stay in similar (or even the same) hotel rooms on each trip. But even if I stay in the same hotel room on subsequent visits, I can't leave something for myself there. Every time that I check into the hotel, I get what is for all intents and purposes a brand new room.

Contrast this with a long-term storage rental. If I rent long-term storage space, I can leave something there on one visit and retrieve it the next time that I return. Even if I leave the city and return again later, the storage locker is mine and what I leave there persists from one visit to the next. When I'm in Seattle, the things I left in my rental storage in Austin are still there. When I get back to Austin, I can go to my storage space and get the things I left there. This is just like a field: the object and its fields continue to exist even when your attention is (temporarily) elsewhere, i.e., even when none of the object's methods are being executed.

The storage locker story is actually somewhat more complex than that, and so is the field story. It might be useful for someone else to have a key to my storage locker, and it is possible for that person to go to Austin and change what's in the locker. So if I share this locker with someone else, what I leave there might not be what I find when I return. It is important to understand that this is still not the same as the hotel room. Between my visits, the hotel cleans out the room. If I leave something in my hotel room, it won't be there the next time I come back. Each time, my hotel room starts out "like new". In contrast, the contents of my storage locker might change, but that is because my locker partner might change it, not because I get a freshly cleaned locker each time that I visit.

The locker partner story corresponds closely to something that can happen with fields. It is possible for the value of a field to change between invocations of the owning object's methods, essentially through the same mechanism (sharing) as the storage locker. To minimize this (when it is not desired), fields are typically declared `private`. For more on this matter, see the discussion of Public and Private in the next chapter. We will return to the issue of shared state (e.g. when two or more people have access to the same airport locker) in the chapter on Synchronization.

#### Whose Data Member is it?

A second way in which fields differ from variables is that every field belongs to some object. For example, in the `Rectangle` code, there's no such thing as `width` in the abstract. Every `width` field belongs to some *particular* `Rectangle` instance, i.e., some object made from the `Rectangle` class/factory/recipe.

Because a field belongs to an object, it isn't really appropriate to refer to it without saying *whose* field you are referring to. Many times, this is easy: `myRectangle.width`, for example, if you happen to have a `Rectangle` named `myRectangle`. The syntax for a field access expression is (1) an object-identifying expression (often, but not always, a name associated with the object), followed by (2) a period, followed by (3) the name of the field. You can now use this as you would any other name:

```
        myRectangle.width = myRectangle.width * 2;
```

for example.

There is, however, a common case in which the answer to the question "whose field is it?" may be an object whose name you don't know. This occurs when you are in a class definition and you want to refer to the instance whose code you are now writing. (Since a class is the set of instructions for how to create an instance, it is common to say "the way to do this is to use my own `width` field....")

In Java, the way to say "myself" is `this`. That is, `this` is a special name expression that is always bound to the current object, the object inside whose code the name `this` appears. That means that the way to say "my own `width` field...." is `this.width`. (Note the period between `this` and `width` -- it is important!)

#### Scoping of Fields

The final way in which fields differ from (local) variables is in their scoping. The scope of a name refers to the segment of code in which that name has meaning, i.e., is a legitimate shoebox or label. (If you refer to a name outside of its scope, your Java program will not compile because the compiler will not be able to figure out what you mean by that name.) A local variable only has scope from its declaration to the end of the enclosing block. (A method's parameter has scope throughout the body of that method.)

A field name has scope anywhere within the enclosing class body. That means that you can use the field name in any other field definition, method body, or constructor body throughout the class, including the part of the class body that is textually prior to the field declaration! For example, the following is legal, if lousy, Java code:

```
class Square{

     int height = this.width;
     int width = 100;
     ...

}
```

(This isn't very good code because (a) it's convoluted and (b) it doesn't do what you think it does. Although `this.width` is a legal expression at the point where it's used, the value of `this.width` is not yet set to 100. The result of this code is to set `height` to 0 and `width` to 100. The rule is: all fields come into existence simultaneously, but their initialization is done in the order they appear in the class definition text.)

A cleaner version of this code would say

```
class Square{

     int height = 100;
     int width = this.height;
     ...

}
```

**Comparison of Kinds of Names**

| | **Class or Interface Name** | **Field (Data Member)** | **Parameter** | **(Local) Variable** |
|---|---|---|---|---|
| **Scope** | Everywhere within containing program or package. | Everywhere within containing class. | Everywhere within method body. | From declaration to end of enclosing block. |
| **Lifetime** | Until program execution completes. | Lifetime of object whose field it is. | Until method invocation completes. | Until enclosing block exits. |
| | | Label names: `null` | Value of matching argument expression supplied to method invocation. | Illegal to use without explicit initialization. |

Shoebox names: value depends on type.

[Footnote: The column for Class or Interface Name refers only to top-level (non-inner) classes or interfaces. The scope and lifetime of an inner class is determined by the context of its declaration.]

**Static Members**

So far, we've said that fields belong to instances made from classes and that each instance made from the class gets its own copy. Recall that the class itself is an object, albeit a fairly different kind of object. (The class is like a factory or a recipe; it is an instance of the class called `Class`.) Sometimes, it is useful for the class object itself to have a field. For example, this field could keep track of how many instances of the class had been created. Every time a new instance was made, this field would be incremented. Such a field would certainly be a property of the class (i.e., of the factory), not of any particular instance of that class.

The declaration for a class object field looks almost like an instance field. The only difference is that class field declarations are preceded by the keyword `static`. [Footnote: The choice of the keyword `static`, while understandable in a historic context, strikes us as an unfortunate one as the common associations with the term don't really accord with its usage here. In Java, `static` means "belonging to the class object."] For example:

```
class Widget {

     static int numInstances;

     ...

}
```

In this case, individual `Widget`s do *not* have `numInstances` fields. There is only one `numInstances` field, and it belongs to the factory, not the `Widget`s. To access it, you would say `Widget.numInstances`. In this case, `this.numInstances` is not legal code anywhere within the `Widget` class.

**Style Sidebar**

## Field Documentation

In documenting a field, you need to indicate what that field represents conceptually to the object of which it is a part. In addition, you should answer these questions as appropriate:

- What range of values can this field take on?
- What other values are interdependent with this one? For example, must this field's value always be updated in concert with another field, or must its value remain somehow consistent with another field?
- Are there any "special" values of this field that carry hidden meaning?

- What methods (or constructors) modify this field? Which read this field? What else relies on its value?
- Where does the value of this field come from?
- Can the value of this field change?

## Methods

In a previous chapter, we saw how method signatures describe the name, parameters, and return type of a method. A method signature declared in an interface ends in a semi-colon; this method specifies a contract, but doesn't say anything about how it works. It is essentially a rule specification. This kind of method -- a specification without an implementation -- is called `abstract`.

Classes specify more than just a contract. Classes also specify how their instances work. In order for an instance to do be able to do something, its class must give more than the rule specification for its methods. An instance needs the rule body for its methods. Classes must supply bodies for any methods promised by the interfaces that they implement. They may also supply additional methods with their own signatures and bodies.

Methods can be identified by the fact that a method name is always followed by an open parenthesis. (There may then be some arguments or parameters, on which more below; there will always be a matching close parenthesis as well.)

### Method Declaration

A **method definition** also follows the type-of-thing name-of-thing convention, but the type-of-thing is the type that is returned when the method is called. So, for example, the `inside` method in the definition of `Rectangle,` above, returns a `boolean` value:

```
...boolean inside( int x, int y) { ... }
```

Inside the parentheses is the list of parameters to the method: calling `pictureFrame.inside` on a particular `x` and `y` value returns true or false depending on whether the point `(x, y)` is inside `pictureFrame`. (Remember that the `inside` method only exists with reference to a particular `Rectangle` -- it's always *some* object's method!) The list of parameters, like every other declaration, follows the type-of-thing name-of-thing convention. Note, though, that while a regular variable definition can declare multiple names with a single type, in a parameter list each name needs its own type.

A few more notes on methods: If there are no parameters, the method takes no arguments, but it must still be declared and invoked with parentheses: `pictureFrame.isEmpty()`, for example. If there is no return value, the return type of the method is `void`. Finally, inside the body of the method, the parameters may be referred to by the names they're given in the parameter declaration. It doesn't matter what other names they might have had outside of the method body, or what else those parameter names might refer to outside the method body. We'll return to the issue of scoping later.

Recall from previous chapters that the method definition as we've described it so far -- the return type and the parameter list -- is also called the signature of the method. It tells you what types of arguments need to be supplied when the method is called -- it must be possible to assign a value of the argument type to a variable of the parameter type -- and what type of thing will be returned when the method is invoked. It doesn't tell you much about the relationships between the method's inputs and its outputs, though. (The method's documentation ought to do that!)

**Style Sidebar**

## Method Implementation Documentation

Documentation for methods in classes is much like the documentation for methods in interfaces. However, class/object methods have bodies as well as signatures. In addition to the usual documentation of the method signature (see the Style Sidebar on Method Documentation in the chapter on Interfaces), your method documentation here should include

- ways in which this method implementation differs from or specializes the documented interface method (signature).
- information concerning the design rationale (why the method works the way that it does), just as you would for any piece of Java code. For more detail, see the Style Sidebar on Documentation in the chapter on Statements.

### Method Body and Behavior

This relationship -- how to get from the information supplied as arguments to the result, or return value -- is the "how to do it" part of the method. Its details are contained in the method body, which -- like a class body -- goes between a pair of braces. What goes in here can be variable definitions or method invocations or any of the complex statements that you will learn about later. You cannot, however, declare other methods inside the body of a method. Instead, the method body simply contains a sequence of instructions that describe how to get from its inputs (if any) to its output (if any), or what else should happen in between.

The body of a method is inside the scope of its parameters. That is, the parameter names may be used anywhere within the method to refer to the corresponding arguments supplied at method invocation time. The body of an instance method is also within the scope of the special name `this`. Just as in fields, inside a method the name `this` refers to the particular instance whose method this is. Static methods -- methods belonging to the class -- are not within the scope of `this`, though. That is, you can't use the special name expression `this` in a static method.

In order to return a value from a method, you use a special statement: `return`. There are actually two forms of this statement: `return(...)`; returns a value (whatever is in the parentheses) from a method invocation. For example,

```
        return (total + 1);
```

returns one more than the value of `total`, though it doesn't change the value of `total` at all. The parentheses around the expression whose value is to be returned are in fact optional, leading to the second form of return: `return;` is used to exit from a method whose return type is `void`, i.e., that does not return anything.

Remember (from the chapter on Expressions) that a method invocation is an expression whose type is the return type of the method and whose value is the value returned by the method. You make this happen (when you're describing the method rule) by using an explicit `return` statement in a method's body. In the chapter on Statements, we saw the execution rule for a method body and how it relates to the evaluation rule for method invocation. This process is summarized in the sidebar on Method Invocation and Execution.

**A Method ALWAYS Belongs to an Object**

A method is a thing that can be *done* (or **invoked**, or **called**). For example, a painting program can draw a line, so `drawLine` could be the name of a method. ***Every method belongs to a particular object.*** For instance, each `increment` method belongs to a particular `ScoreCounter` (or `Stopwatch`, or...) object; there is no such thing as an independent `getValue` method. So, if `myScoreCounter` refers to a particular `ScoreCounter`, `myScoreCounter.getValue()` invokes `myScoreCounter`'s `int`-returning method. You can't just call `getValue()`. Whose `getValue()` method is it, anyway?

Each time that you refer to a method, you should ask yourself whose method it is. You can invoke a method by first referring to the object, then typing a period, then the method name, as in `myScoreCounter.getValue()`. Sometimes, the answer to "whose method is it?" will be "my own", that is, the method belongs to the object whose code is being executed. As with fields, the way to say "myself" is with the special name expression `this`, so the way to say "my `getValue()` method" is `this.getValue()`. (Note the period between `this` and `getValue()` -- it is important!)

Generally, methods belong to instances of the class in which they're defined. Occasionally, though, it may be useful to have a method that belongs to the class itself. This corresponds to a property of the factory (or recipe), rather than one belonging to the widgets (or cookies) produced. For example, a method that prints out the number of widgets produced by the factory so far would be a method belonging to the factory, not one belonging to any particular widget. Methods that belong to the class instead of to its instances look just like regular methods, except that they are prefaced with the keyword `static`. (This name is pretty unintuitive, though it makes some sense in its historical context. Remember: In Java, `static` means "belonging to the class/factory/recipe itself, not to its instances.") A static method can be addressed by first citing the object it belongs to, then period, then the method name: `Widget.howManyWidgets()`. A static method *should not* be invoked using `this`, though, because it doesn't belong to an instance.

Inside the method body, the name `this` may be treated as any other name.

it is also possible to refer to the object whose method it is as (for example, if you want to pass it as an argument to another method).

**Method Overloading**

Just as in an interface, it is possible for a class to have multiple methods with the same name. This is called **method overloading**, since the name of the method is overloaded -- it actually refers to two or more distinct methods -- belonging to that object. In this case, each method must have a different footprint, i.e., the ordered list of parameter types must differ for two methods of the same object with the same name.

When an object has an overloaded method, the particular method to be invoked is selected by comparing the types of the arguments supplied with the footprints of the methods. The method whose footprints best matches the (declared) types of the arguments supplied is the one that is invoked. This matching is done using the same type inclusion rules as the operator `instanceof`.

## Method Invocation and Execution

Method invocation is an expression; it is evaluated, producing a value. Within this expression, the body of the method is treated as a block (sequence) statement to be executed. This sidebar summarizes this process.

1. Before the method invocation expression can be evaluated, the object expression describing whose method it is must be evaluated. This object is called the method's **target**.
2. Based on this object and the (declared) types of the argument expressions, the method body is selected.
3. The argument expressions are evaluated and the method parameter names are bound to the corresponding arguments. If the target is an instance (i.e., if the method is not static), the name `this` is bound to the target as well.
4. Within the scope of these name bindings, the body of the statement is executed as a normal block except for special rules concerning `return` statements.
   - If, at any point within the execution of the body, a return statement is encountered, its expression (if present) is evaluated and then the entire method body and the scope of parameter names and `this` are exited upon completion of the return statement.
   - If the method has a return type other than void, the return statement is mandatory and must include an expression whose type is consistent with the return type. A suitable return statement must be encountered on any normal execution path through the method body. In this case, the value of the return expression is the value returned by the method invocation expression.
   - If the return type of the method is void, the final closing brace of the method body is treated as an implicit return; statement, i.e., a return with no expression. This has the effect of exiting the method body and special name scope.

### Constructors

So, how do objects get created? Each class has a special member, called a **constructor**, which gives the instructions needed to create a new instance of the class. (If you don't give your class a constructor, Java automatically uses a default constructor, which roughly speaking "just creates the instance" -- details below. So some of the classes that you see may not appear to have constructors -- but they all do.)

**Constructor are Not Methods**

A constructor is sort-of like a method.

1. It has a (possibly empty) parameter list enclosed in parentheses.
2. It has a body, enclosed in braces, consisting of statements to be executed.
3. Inside the constructor body, `this.` expressions can be used to refer to methods and fields of the individual instance under construction.

There are several differences.

1. The name of a constructor always matches the name of the class whose instances it constructs.
2. A constructor has no return type.
3. A constructor does not return anything; `return` statements are not permitted in constructors.
4. A constructor cannot be invoked directly.

Instead, a constructor is invoked as a part of a `new` expression. The result of evaluating this `new` expression is a new instance of the type whose constructor is evoked.

For example:

```
class Pie {
```

might have the constructor

```
     Pie (Ingredients stuff) {
       stuff.bake();
     }

}
```

In other words, to create a `Pie`, bake its ingredients. Note that `stuff` is a parameter, just like in a method. Constructor parameters work exactly like method parameters, and constructors take arguments to match these parameters in the same way that methods take parameters.

But you don't invoke a constructor in the same way that you invoke a method. In order to invoke a method, you need to know whose method it is. In order to use a constructor, you only need to know the name (and parameter type list) of the constructor. You invoke a constructor with a new expression as follows:

```
new Pie ( myIngredients )
```

where `myIngredients` is of type `Ingredients`.

**Syntax**

The syntax of a constructor is similar to, but not identical to, the syntax of a method. A constructor may begin with a visibility modifier (i.e., `public`, `protected`, or `private`) or one of a handful of other modifiers. Next comes the name of the constructor, which is always identical to the name of the class. The name is followed by a comma-separated parameter list enclosed in parentheses. This parameter list, like the parameter list of a method, consists of type-of-thing name-of-thing pairs. As in a method, the constructor name plus the ordered list of parameter types forms the constructor's footprint. It is possible for a class to have multiple constructors as long as they have distinct footprints.

After the parameter list, a constructor has a body enclosed in braces. This body is identical to a method body -- an arbitrary sequence of statements -- except that it may not contain a `return` statement. This is because constructors are not methods that can be called and that return values of specified types; instead, a constructor is invoked using a `new` expression whose value is a new instance of the constructor class's type. The constructor body may contain any other kind of expression or statement, however, including declarations or definitions of local variables.

```
modifiers ClassName ( type_1 name_1, ... type_n name_n )
{
    // body statements go here
}
```

For example, the NameDropper StringTransformer class might begin as follows. Note that the constructor argument is used to initialize the private field, the particular name that *this* NameDropper will drop.

```
public class NameDropper extends StringTransformer
{

     private String who;

     public NameDropper ( String name )
     {
         this.who = name;
     }

     //etc.
```

Note the use of a `this.` expression to refer to the field of the particular NameDropper instance being created.

This constructor could be invoked using the expression `new NameDropper( "Jean" )` or `new NameDropper( "Terry" )`.

[Pictures of NameDroppers Jean and Terry]

**Style Sidebar**

## Constructor Documentation

Although a constructor is not a method, documentation for a constructor is almost identical to documentation for a method. Constructor documentation should include:

- specifics distinguishing this constructor from others

- preconditions for using this constructor
- parameters required and their role(s)
- relationship of the constructed object to parameters or other factors
- side effects of the constructor
- additional assumptions and design rationale as appropriate

**Execution Sequence**

Before a constructor is invoked, the instance is actually created. In particular, any shoeboxes or labels declared as fields of the instance are created before the execution of any constructor code. This permits access to these fields from within the constructor body. In addition, any initialization of these fields -- through definitions in their declarations -- is executed at this time as well. Fields are each created and then each initialized in textual order, but all fields -- even those declared after the constructor[Footnote: There should be no such fields, declared after the constructor, because this makes your code difficult to read and so is bad style. However, if any such declarations are made, they still executed prior to the constructor itself.] -- are created and initialized prior to the execution of the constructor. Once each of the instance fields is created, execution of the constructor itself can begin.

When a constructor is executed, its parameters are matched with the arguments supplied in the invocation (`new`) expression. For example, in the body of the NameDropper constructor, the name `name` is identified with the particular String supplied to the constructor invocation expression. So if the constructor were invoked with the expression `new NameDropper( "Terry" )`, the name `name` would be associated with the String "Terry" during the execution of the body of the NameDropper constructor. When the statement

```
this.who = name;
```

is executed, the value of the expression `name` is the String "Terry".

Once each of the parameter names has been associated with the corresponding argument, the execution of the statements constituting the constructor's body proceeds in order (except where that order is modified by control-flow expressions such as `if` or `while`). These statements may include local variable declarations; in this case, the name declared has scope from its declaration to the end of the enclosing block, just as in a method. When the end of the constructor is reached, execution of the constructor invocation expression is complete and the value -- the new instance -- is produced.

Because a constructor body may not contain a return statement, it is not possible to exit normally from any part of the constructor body except the end. Judicious use of conditionals can simulate this effect, however.

**Multiple Constructors and the Implicit No-Arg Constructor**

A class may have more than one constructor as long as each constructor has a different footprint, i.e., as long as they have different ordered lists of parameter types. So, for example, NameDropper might also have a variant constructor that took a descriptive phrase as well as name:

```
          public NameDropper ( String name, String adjective )
          {
              this.who = adjective + " " + name;
          }
```

In this case, `new NameDropper( "Marilyn Monroe" )` would create a NameDropper that started every phrase with "Marilyn Monroe says..." while

```
     new NameDropper( "Norma Jean", "My dear friend" )
```

(i.e., `NameDropper(String, String)` )would attribute everything to "My dear friend Norma Jean..."

If -- and only if -- a class contains no constructors at all, a default constructor is assumed present. This default constructor takes no arguments and does nothing beyond creating the object (and initializing the fields if they are defined in their declarations).

If there is even one constructor, the implicit no-arg constructor is not assumed. This means that if you define a constructor such as the one for NameDropper, above, that takes a parameter, the class will not have a no-arg constructor (unless you define one).

Beware: This can cause a problem when extending a class, if you're not careful. See chapter on Inheritance.

**Constructor Functions**

Often, one of the main functions of a constructor is to initialize the state of the instance you're creating. Some initializations don't require a constructor; they can happen when the field is declared, by using a definition instead of a simple declaration:

```
     class LightSwitch {

          boolean isOn = false;

     }
```

In this case, each `LightSwitch` instance is created in the off position. In this kind of initialization, each instance of the class has its field created with the same initial value.

Contrast this with the following example, in which the initial value of the `name` field isn't known until the particular `Student` instance is created.

```
     class Student {

          String name;

          Student( String who ) {
            this.name = who;
          }

     }
```

In this case, a constructor is used to explicitly initialize the field named `name`. When the initial value of a field varies from instance to instance, it cannot be assigned in the field declaration. Instead, it must be assigned at the time that the particular instance is created: in the constructor.

A constructor (or a method body) can also refer to properties of the class object itself. Recall the Widget class, which kept track of how many instances had been created. When the constructor is invoked, it can increment the appropriate field:

```
class Widget {

     static int numInstances;

     static int howManyWidgets(){
         return Widget.numInstances;
     }

     Widget(){
         Widget.numInstances = Widget.numInstances + 1;
     }

}
```

Note that the constructor is not declared `static` (Constructors don't properly belong to any object) but that it refers to a `static` field. Note also that the `static` field is referred to using the class name (`Widget`), *not* using `this`. We've also filled in the static method referred to above.

Finally, note that there is no explicit return statement in a constructor. A constructor is not a method, and it cannot be invoked directly. Instead, it is used in a construction expression, with the keyword `new`: `new Widget()` is an expression whose type is `Widget` and whose value is a brand new instance of the `Widget` class, for example.

**Q.** Construct a `Counter` class which supports an increment (increase-by-one) method. Where does the `Counter`'s initial value come from?

**Style Sidebar**

## Capitalization Conventions

By convention, the first letters of all class and interface names are capitalized. Since constructor names match their classes, constructor names also begin with capital letters. Java file names also match the class (or interface) declared within, so Java file names begin with a capital letter.

All other names (except constants) begin with lower case letters. In particular, the names of Java primitive types begin with lower case letters, as do fields, methods, variables, and parameters.

After the first letter, mixed case is used, with subsequent capital letters indicating the beginnings of intermediate words: e.g., *`ClassName`* `and` *`instanceName`*.

The exception to the above conventions is the capitalization of constants (i.e., static final fields; see below). The names of constants are entirely capitalized. Intermediate words are separated using underscores (_): `CONSTANT_NAME`.

## Summary

- A Java class is a Java type.
- Each (public, top level) class must be defined in a separate file whose name matches the class name.
- An instance of a class is an object whose type is that class.
- If a class implements an interface, its instances must satisfy the interface's promises.
- Classes have methods, fields, and constructors.
- In a class, methods typically have bodies specifying how to carry out the method. (Otherwise, the method is `abstract`, and so is the class.)
- Every method belongs to some object. Unless declared `static`, a method belongs to (each of) a class's instances, not to the class itself.
- A field declares (and perhaps also defines) a name whose scope is the class body (i.e., any methods, fields, or constructors in the class body) and whose lifetime is the lifetime of the instance it belongs to.
- Every field belongs to some object. Unless declared `static`, a field belongs to (each of) a class's instances. Each instance has its own copy of the field, i.e., its own unique label with that field's name and type.
- In Java, `this` is a special name, bound in any non-static member, that refers to the instance whose instructions are being followed. An instance can refer to its own methods and fields by saying `this.`*`methodName`*`(...)` or `this.`*`fieldName`*, or to itself by the name expression `this`.
- A constructor gives instructions for how to create an instance of the class.
- The class itself is an object. (It is an instance of the class `Class`.) Fields and methods declared `static` belong to the class object itself and are properly referred to using *`ClassName`*`.`*`methodName`*`(...)` or *`ClassName`*`.`*`fieldName`*.

## Exercises

1. Consider the following definition:

```
public class MeeterGreeter
{

     private String greeterName;

     public MeeterGreeter( String name )
     {
         this.greeterName = name;
```

```
        }

        public void sayHello()
        {
            Console.println( "Hello, I'm " + this.greeterName );
        }

        public void sayHello( String toWhom )
        {
            Console.println( "Hello, " + toWhom
                                  + ", I'm " + this.greeterName );
        }

        public String getNameWithIntroduction ( String toWhom )
        {
            // ****
            this.sayHello( toWhom );
            return this.greeterName;
        }

    }
```

Now assume that the following definition is executed:

```
    MeeterGreeter pat = new MeeterGreeter( "Pat" ),
                  terry = new MeeterGreeter( "Terry" );
```

a. What is printed by `pat.sayHello()`? What is returned? Which method is invoked?
2. What is printed by `new MeeterGreeter( "Chris" ).sayHello( "Terry" )`? What is returned? Which method is invoked?
3. What is printed by `terry.sayHello( "Pat" )`? What is returned? Which method is invoked?
4. Assume that the expression `pat.getNameWithIntroduction( "Chris" )` is being evaluated. What would the value be of each of the following expressions if they were to appear on the ****'d line:
    i. `toWhom`
    2. `this.greeterName`
    3. `name`
    4. `this.sayHello()`
    5. `new MeeterGreeter( "Pat" )`
    6. `this.getNameWithIntroduction( toWhom );`

2. Now consider the following modification of the `MeeterGreeter` code. Assume that we add the field definition

```
        private static String greeting = "Hello";
```

We will want to make several other modifications to the `MeeterGreeter` code.

a. Write a `changeGreeting` method that allows a user to change the greeting string.

        i. What arguments should this take?
        2. What should it return?
        3. What should its body say?
        4. To which object should this method belong?
    ii. Write an expression that invokes the `changeGreeting` method that you have written.
    3. Next, modify the sayHello methods to replace the fixed string "Hello" with the a reference to the greeting field. Whose greeting field is it?

3. Define a class whose instances each have one method, `rememberAndReturnPrevious`, that takes a String and returns the String it was previously given. Supply the first return value through the instance creation expression. Give an example of your code in use.