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 sciences building</td>
</tr>
<tr>
<td>rectangle</td>
<td></td>
</tr>
</tbody>
</table>
Objects are collections of data and sets of operations
Objects are collections of *data* and sets of *operations*
Objects are collections of *data* and sets of *operations*.

<table>
<thead>
<tr>
<th>CartoonPlutocrat</th>
</tr>
</thead>
<tbody>
<tr>
<td>- name</td>
</tr>
<tr>
<td>- bankBalance</td>
</tr>
<tr>
<td>- earnMoney(int)</td>
</tr>
<tr>
<td>- spendMoney(int)</td>
</tr>
<tr>
<td>- getBalance()</td>
</tr>
</tbody>
</table>

(all trademarks are property of their respective owners)
Objects are collections of *data* and sets of *operations*
Objects are collections of *data* and sets of *operations*.

- **name**
- **bankBalance**
- **earnMoney(int)**
- **spendMoney(int)**
- **getBalance()**
Objects are collections of data and sets of operations
**Accessor methods** inspect the data of an object

<table>
<thead>
<tr>
<th>CartoonPlutocrat</th>
</tr>
</thead>
<tbody>
<tr>
<td>- name</td>
</tr>
<tr>
<td>- bankBalance</td>
</tr>
<tr>
<td>- earnMoney(int)</td>
</tr>
<tr>
<td>- spendMoney(int)</td>
</tr>
<tr>
<td>- getBalance()</td>
</tr>
</tbody>
</table>
**Accessor methods** inspect the data of an object

```plaintext
CartoonPlutocrat

- name
- bankBalance
- earnMoney(int)
- spendMoney(int)
- getBalance()
```

**FIELDS (DATA)**
**Accessor methods** inspect the data of an object

<table>
<thead>
<tr>
<th>CartoonPlutocrat</th>
</tr>
</thead>
<tbody>
<tr>
<td>- name</td>
</tr>
<tr>
<td>- bankBalance</td>
</tr>
<tr>
<td>- earnMoney(int)</td>
</tr>
<tr>
<td>- spendMoney(int)</td>
</tr>
<tr>
<td>- getBalance()</td>
</tr>
</tbody>
</table>
Accessor methods inspect the data of an object

Mutator methods modify the data of an object
**Accessor methods** inspect the data of an object

<table>
<thead>
<tr>
<th>CartoonPlutocrat</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name</td>
</tr>
<tr>
<td>-bankBalance</td>
</tr>
<tr>
<td>-earnMoney(int)</td>
</tr>
<tr>
<td>-spendMoney(int)</td>
</tr>
<tr>
<td>-getBalance()</td>
</tr>
</tbody>
</table>

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

```java
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
Classes are "units" of code containing methods

```java
public class Hello {
}
```
Methods are sequences of simple instructions

```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 are like sentences: single, imperative instructions

```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);
    }
}
```
Let’s talk about statements a bit.
Statements are made up of identifiers and operators.

String greeting = "Hello, world!";
Statements are made up of identifiers and operators.

String greeting = "Hello, world!";
Statements are made up of identifiers and operators.

```java
String greeting = "Hello, world!";
```
Statements are made up of identifiers and operators.

```java
String greeting = "Hello, world!";
```
Statements are made up of identifiers and operators.

```java
String greeting = "Hello, world!";
```
Statements are made up of identifiers and operators.

String greeting = "Hello, world!";
(We’ll talk more about identifiers later)
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, barney, FRED, f123, fredFlintstone, __why_s0_many_underscores

• Not legal
  • class, 2legit2quit, blah!, dot.com, #$!@
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.
Classes, methods, and statements comprise the anatomy of a program

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!";
Read this as `greeting gets the literal string “Hello, World”`

String greeting = "Hello, world!";

(We pronounce = “gets”)
Declaring variables

- **Declaration**: `type id;
  - String foo;
  - int x;
  - Clicker y;

- **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.
String s = "Hello!";
String t = s;

t and s 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);

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

What happens now?
String s = "Hello, world!";
String t = s;
t = "Hello, friends!";
String s = "Hello, world!";
String t = s;
t = "Hello, friends!";
String s = "Hello, world!";
String t = s;
t = "Hello, friends!";
Consider the difference between putting a business card in your wallet and putting an office in your wallet!
Clicker c = new Clicker();
Clicker d = c;

d.click();
System.out.println(c.getCticks());

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”;`
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 \textit{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);
    }
}
```
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 Hello2 {
    public static void main(String[] args) {
        String greeting = "Howdy!";
        {
            String greeting = "Hello!";
            System.out.println(greeting);
        }
    }
}
```
What about this...

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

The scope for `main` encloses the blue scope!
...or this?

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

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

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

There is no greeting in scope at this point.
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
Usually, example programs involve mundane business problems
Usually, example programs involve mundane business problems
Usually, example programs involve mundane business problems.
What objects are involved in a life-and-death problem?
What objects are involved in a life-and-death problem?
What objects are involved in a life-and-death problem?
What objects are involved in a life-and-death problem?
What objects are involved? What data and methods should each object include?
What objects are involved? What data and methods should each object include?
What objects are involved? What data and methods should each object include?
What objects are involved? What data and methods should each object include?
What objects are involved? What data and methods should each object include?
What does the player need to know?

• score
• “lives”
• position
• how to redraw
• how to move
What does the foe need to know?

• position
• whether to chase, flee, or return
• how to redraw
• how to move
OK, we've got a better idea of how to think about objects. How can we use objects?
Objects

• Recall that an \textit{object} is a collection of fields (viz., data) and \textit{methods} that access and operate on that data

• We’ve already seen some objects in class:
  • System.out
  • greeting

• \textbf{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 operator, 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 = 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!

```java
public class Clicker {
    public void click();
    public int inspect();
    public void reset();
}
```
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;
}
public int inspect() {
    return clicks;
}

public int inspect() {
    return clicks;
}
Accessor methods inspect an object’s state.
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 invoke a “ctor” directly; Java does when you 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
}
```
Example ctors

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

    public Clicker() {
        clicks = 0;
    }

    public Clicker(int clicks) {
        this.clicks = clicks;
    }

    // ...other methods
}
```

What is the value of `clicks` before this statement executes?
Example ctors

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

    public Clicker() {
        clicks = 0;
    }

    public Clicker(int clicks) {
        this.clicks = clicks;
    }

    // ...other methods
}
```

**What is the value of CLICKS before this statement executes?**

**What do we suppose that this statement does?**

**And what does "this" mean?**
Just to make sure you’re paying attention: What happens when you create an object?
Dog otto = new Dog("Otto", "otto.jpg");
Creating an object

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

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

1. The Java VM sets aside a Dog-shaped chunk of RAM.
Creating an object

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

2. The JVM initializes the fields of the new Dog to the default values for each type.
Creating an object

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

3. The JVM executes user constructor code to fill in the values for each field.
Creating an object

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

3. The JVM executes user constructor code to fill in the values for each field.

(This may include constructing additional objects!)

<table>
<thead>
<tr>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Otto&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dog</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>image</td>
</tr>
</tbody>
</table>
Creating an object

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

3. The JVM executes user constructor code to fill in the values for each field.

(This may include constructing additional objects!)
Creating an object

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

3. The JVM executes user constructor code to fill in the values for each field.

(This may include constructing additional objects!)
Creating an object

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

3. The JVM executes user constructor code to fill in the values for each field.

(This may include constructing additional objects!)
Creating an object

Dog otto = new Dog(“Otto”, “otto.jpg”);

3. The JVM executes user constructor code to fill in the values for each field.

(This may include constructing additional objects!)
Creating an object

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

4. Finally, the JVM gives otto a reference to the newly-allocated Dog.
Creating an object

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

4. Finally, the JVM gives `otto` a reference to the newly-allocated `Dog`.

```
<table>
<thead>
<tr>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Otto&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dog</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>image</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Image</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
```
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.
Thing a, b, c;
Thing x = new Thing();
Thing y = new Thing();
Thing z = new Thing();
/* point 1 */

a = x;
b = a;
c = y;
/* point 2 */
a = y;
c = z;
/* point 3 */
y = z;
/* point 4 */
a = b;
/* point 5 */
return;
/* point 6 */
Thing a, b, c;
Thing x = new Thing();
Thing y = new Thing();
Thing z = new Thing();
/* point 1 */

a = x;
b = a;
c = y;
/* point 2 */

a = y;
c = z;
/* point 3 */

y = z;
/* point 4 */

a = b;
/* point 5 */

return;
/* point 6 */
Thing a, b, c;
Thing x = new Thing();
Thing y = new Thing();
Thing z = new Thing();
/* point 1 */

a = x;
b = a;
c = y;
/* point 2 */

a = y;
c = z;
/* point 3 */

y = z;
/* point 4 */

a = b;
/* point 5 */

return;
/* point 6 */
Thing a, b, c;
Thing x = new Thing();
Thing y = new Thing();
Thing z = new Thing();
/* point 1 */

a = x;
b = a;
c = y;
/* point 2 */

a = y;
c = z;
/* point 3 */
y = z;
/* point 4 */

a = b;
/* point 5 */

return;
/* point 6 */
Thing a, b, c;
Thing x = new Thing();
Thing y = new Thing();
Thing z = new Thing();
/* point 1 */

    a = x;
    b = a;
    c = y;
    /* point 2 */

    a = y;
    c = z;
    /* point 3 */

    y = z;
    /* point 4 */

    a = b;
    /* point 5 */

return;
/* point 6 */
Thing a, b, c;
Thing x = new Thing();
Thing y = new Thing();
Thing z = new Thing();
/* point 1 */

a = x;
b = a;
c = y;
/* point 2 */

a = y;
c = z;
/* point 3 */

y = z;
/* point 4 */

a = b;
/* point 5 */

return;
/* point 6 */
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 parameters are omitted for clarity).
Sidebar: \texttt{methods}

```java
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;
}
```
Sidebar: methods

```java
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;
}

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;
}

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;
}

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;
}
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;
}
sidebar: methods

```java
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;
}
```

```
| fred(int) | i | 4 |
| current   | x | 20 | y |
|           |   |    | z |

int x = 5 * i;
```
Sidebar: methods

```java
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;
}
```

<table>
<thead>
<tr>
<th>fred(int)</th>
<th>i</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>current</td>
<td></td>
<td></td>
</tr>
<tr>
<td>z</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

int y = 4 * i;
Sidebar: methods

```java
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;
}
```

| fred(int) | i | 4 |
| current   | x | 20 |
|           | y | 16 |
|           | z |    |
Sidebar: methods

```java
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;
}
```

<table>
<thead>
<tr>
<th>fred(int)</th>
<th>i</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

return barney(x + y);
Sidebar: methods

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;
}
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;
}
Sidebar: methods

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;
}
Sidebar: methods

```java
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;
}
```

Copyright © 2005-2007 William C. Benton
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;
}
Sidebar: methods

```java
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;
}
```

```
<table>
<thead>
<tr>
<th>fred(int)</th>
<th>i</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>current</th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
</tr>
</tbody>
</table>

int x = i % 8;
```
Sidebar: methods

```java
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;
}
```

Copyright © 2005-2007 William C. Benton
Sidebar: methods

```java
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;
}
```

Copyright © 2005-2007 William C. Benton
A method name is *overloaded* if a class has more than one method with the same name but different parameter types.
The Java compiler selects the appropriate overloaded method based on the method signature.
Example

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

public void click(int x) {
    clicks = clicks + x;
}
Example

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

public void click(int x) {
    clicks = clicks + x;
}
```

note different parameter list!
public class Clicker {
    private int clicks;

    public Clicker() {
        clicks = 0;
    }

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

    public void click(int x) {
        clicks = clicks + x;
    }

    public void click(float x) {
        clicks =
        clicks + (int)x;
    }

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

Clicker c = new Clicker();
c.click();
c.click(5);
c.click();
c.click(6);
c.click(5.0f);
Organizing classes

• How do we make sure that every class has a unique name?
  • e.g. my Dog class vs. your Dog class

• One solution: add unique “tag” to each class name identifier
  • e.g. MyDog vs. YourDog

• Will this work? Why or why not?
Packages provide a way to organize classes in a hierarchical namespace.
Hierarchical namespaces

• We use these in real life all the time. For example:
  • Your friend Abe
  • Abraham Lincoln
  • Abraham Lincoln of Springfield, IL
• More examples from local news anchors:
  • “former Badger Chris Chelios”
  • “UW-Superior grad Arnold Schwarzenegger”
Packages in practice

• Classes are organized into packages
  • java.util.HashMap
  • edu.wisc.cs.willb.PRNGenerator

• Can refer to fully-qualified class name (i.e. package+class)

• Can also import one or several classes into namespace
import

• The import keyword is a way to say: in this file, when I say C, I mean p.C.

• import java.util.HashMap;

• All subsequent references to HashMap in that file mean java.util.HashMap.

• Imports come at the beginning of a file.

• Import entire packages:
  import java.util.*;
There are *three kinds of places* to store values in Java.
Local variables are named places to store values and are particular to a given method invocation.
Parameter variables store copies of the arguments to a given method.
Instance fields store the private data of an object. Think of them as variables that belong to an instance.
Three places

• *Local variables* belong to method invocations

• *Parameter variables* belong to method invocations and are given values by method arguments

• *Instance fields* belong to individual objects
Some review questions
Review: objects and classes

• 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?