## 10 numbers and statics

Numbers Matter
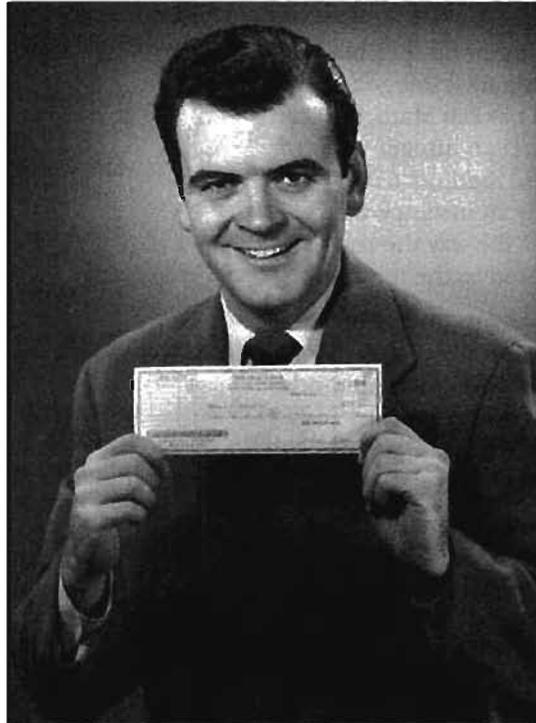
Do the Math. But there's more to working with numbers than just doing primitive arithmetic. You might want to get the absolute value of a number, or round a number, or find the larger of two numbers. You might want your numbers to print with exactly two decimal places, or you might want to put commas into your large numbers to make them easier to read. And what about working with dates? You might want to print dates in a variety of ways, or even *manipulate* dates to say things like, "add three weeks to today's date". And what about parsing a String into a number? Or turning a number into a String? You're in luck. The Java API is full of handy number-tweaking methods ready and easy to use. But most of them are **static**, so we'll start by learning what it means for a variable or method to be static, including constants in Java—static *final* variables.

MATH methods: as close as you'll ever get to a *global* method

Except there's no global *anything* in Java. But think about this: what if you have a method whose behavior doesn't depend on an instance variable value. Take the round() method in the Math class, for example. It does the same thing every time—rounds a floating point number(the argument to the method) to the nearest integer. Every time. If you had 10,000 instances of class Math, and ran the round(42.2) method, you'd get an integer value of 42. Every time. In other words, the method acts on the argument, but is never affected by an instance variable state. The only value that changes the way the round() method runs is the argument passed to the method!

Doesn't it seem like a waste of perfectly good heap space to make an instance of class Math simply to run the round() method? And what about *other* Math methods like min(), which takes two numerical primitives and returns the smaller of the two. Or max(). Or abs(), which returns the absolute value of a number.

These methods never use instance variable values. In fact the Math class doesn't *have* any instance variables. So there's nothing to be gained by making an instance of class Math. So guess what? You don't have to. As a matter of fact, you can't.

Methods in the Math class don't use any instance variable values. And because the methods are 'static', you don't need to have an instance of Math. All you need is the Math class.

```
int x = Math.round(42.2);
int y = Math.min(56,12);
int z = Math.abs(-343);
```

↑

These methods never use instance variables, so their behavior doesn't need to know about a specific object

If you try to make an instance of class Math:

```
Math mathObject = new Math();
```

You'll get this error:

```
File Edit Window Help IwasToldThereWouldBeNoMath
%javac TestMath

TestMath.java:3: Math() has private
access in java.lang.Math

     Math mathObject = new Math();
                       ^

1 error
```

←This error shows that the Math constructor is marked private! That means you can NEVER say 'new' on the Math class to make a new Math object

The difference between regular (non-static) and static methods

Java is object-oriented, but once in a while you have a special case, typically a utility method (like the Math methods), where there is no need to have an instance of the class. The keyword **static** lets a method run *without any instance of the class*. A static method means "behavior not dependent on an instance variable, so no instance/object is required. Just the class."

regular (non-static) method

```
public class Song {
    String title;
    public Song(String t) method.
        title = t;
    }
    public void play() {
        SoundPlayer player = new SoundPlayer();
        player.playSound(title);
    }
}
```
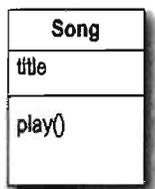
Instance variable value affects the behavior of the play()

The current value of the 'title' instance variable is the song that plays when you call play().

Song

title
play()

two instances of class Song

Politik Coldplay — Song object

*My Way Sex Pistols* — Song object

s2 — Song

s3 — Song

s2.play();

s3.play();

Calling play() on this reference will cause "Politik" to play.

Calling play() on this reference will cause "My Way" to play.
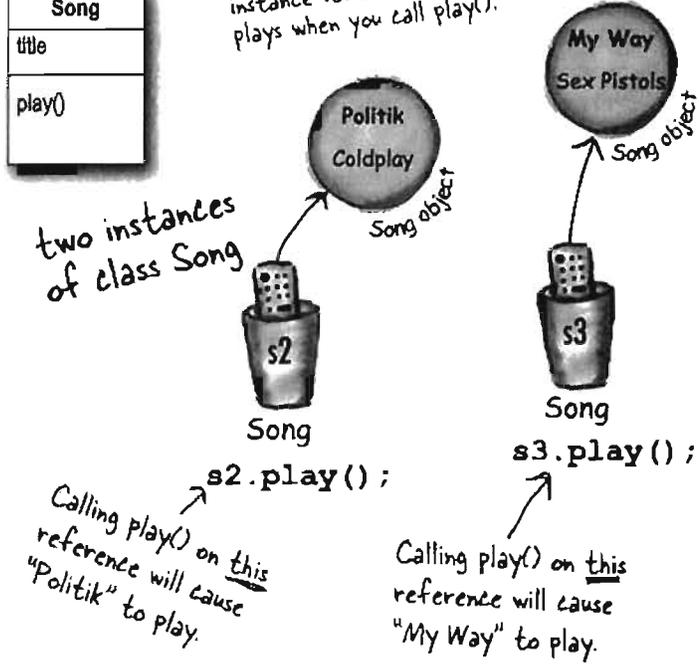
static method

```
public          int min(int a, int b){
    //returns the lesser of a and b
}
```

Math

min()
max()
abs()
...
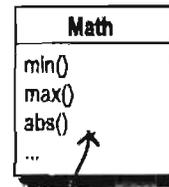
No instance variables. The method behavior doesn't change with instance variable state.

Math.min(42,36);

Use the Class name, rather than a reference variable name.

NO OBJECTS!! Absolutely NO OBJECTS anywhere in this picture!

Call a static method using a class name

Math.min(88,86);
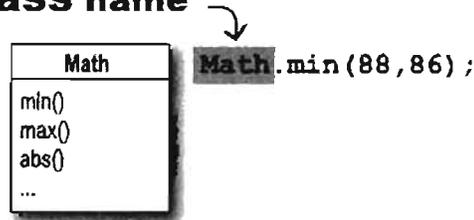
Call a non-static method using a reference variable name

Song t2 = new Song();
t2.play();

What it means to have a class with static methods.

Often (although not always), a class with static methods is not meant to be instantiated. In Chapter 8 we talked about abstract classes, and how marking a class with the **abstract** modifier makes it impossible for anyone to say 'new' on that class type. In other words, *it's impossible to instantiate an abstract class.*

But you can restrict other code from instantiating a *non*-abstract class by marking the constructor **private**. Remember, a *method* marked private means that only code from within the class can invoke the method. A *constructor* marked private means essentially the same thing—only code from within the class can invoke the constructor. Nobody can say 'new' from *outside* the class. That's how it works with the Math class, for example. The constructor is private, you cannot make a new instance of Math. The compiler knows that your code doesn't have access to that private constructor.

This does *not* mean that a class with one or more static methods should never be instantiated. In fact, every class you put a main() method in is a class with a static method in it!

Typically, you make a main() method so that you can launch or test another class, nearly always by instantiating a class in main, and then invoking a method on that new instance.

So you're free to combine static and non-static methods in a class, although even a single non-static method means there must be *some* way to make an instance of the class. The only ways to get a new object are through 'new' or deserialization (or something called the Java Reflection API that we don't go into). No other way. But exactly *who* says new can be an interesting question, and one we'll look at a little later in this chapter.

Static methods can't use non-static (instance) variables!

Static methods run without knowing about any particular instance of the static method's class. And as you saw on the previous pages, there might not even *be* any instances of that class. Since a static method is called using the *class* (*Math*.random()) as opposed to an *instance reference* (*t2*.play()), a static method can't refer to any instance variables of the class. The static method doesn't know *which* instance's variable value to use.

If you try to use an instance variable from inside a static method, the compiler thinks, "I don't know *which* object's instance variable you're talking about!" If you have ten Duck objects on the heap, a static method doesn't know about any of them.

If you try to compile this code:

```java
public class Duck {

    private int size;

    public static void main (String[] args) {
        System.out.println("Size of duck is " + size);
    }

    public void setSize(int s) {
        size = s;
    }
    public int getSize() {
        return size;
    }
}
```

Which Duck? Whose size?

If there's a Duck on the heap somewhere, we don't know about it
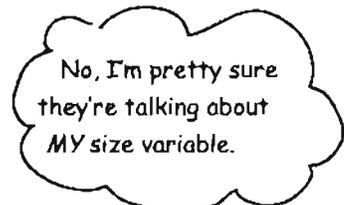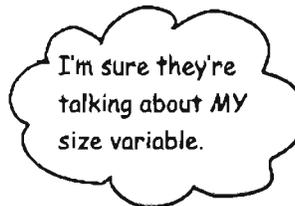
You'll get this error:

```
File Edit Window Help Quack
% javac Duck.java

Duck.java:6: non-static variable
size cannot be referenced from a
static context

        System.out.println("Size
of duck is " + size);

         ^
```

I'm sure they're talking about MY size variable.

No, I'm pretty sure they're talking about MY size variable.

Static methods can't use non-static methods, either!

What do non-static methods do? *They usually use instance variable state to affect the behavior of the method.* A getName() method returns the value of the name variable. Whose name? The object used to invoke the getName() method.

This won't compile:

Calling getSize() just postpones the inevitable—getSize() uses the size instance variable.

```
public class Duck {

    private int size;

    public static void main (String[] args) {
        System.out.println("Size is " + getSize());
    }

    public void setSize(int s) {
        size = s;
    }
    public int getSize() {
        return size;
    }
}
```

Back to the same problem... whose size?

```
File Edit Window Help Jack-in

% javac Duck.java

Duck.java:6: non-static method
getSize() cannot be referenced
from a static context

        System.out.println("Size
of duck is " + getSize());

        ^
```

Make it Stick

Roses are red,
and known to bloom late

Statics can't see
instance variable state
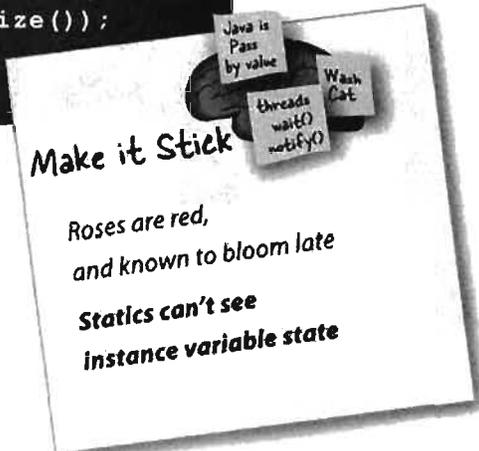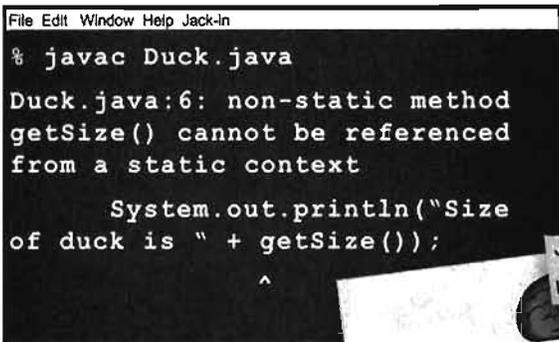
Q: What if you try to call a non-static method from a static method, but the non-static method doesn't use any instance variables. Will the compiler allow that?

A: No. The compiler knows that whether you do or do not use instance variables in a non-static method, you *can*. And think about the implications... if you were allowed to compile a scenario like that, then what happens if in the future you want to change the implementation of that non-static method so that one day it *does* use an instance variable? Or worse, what happens if a subclass *overrides* the method and uses an instance variable in the overriding version?

Q: I could swear I've seen code that calls a static method using a reference variable instead of the class name.

A: You *can* do that, but as your mother always told you, "Just because it's legal doesn't mean it's good." Although it *works* to call a static method using any instance of the class, it makes for misleading (less-readable) code. You *can* say,

```
Duck d = new Duck();
String[] s = {};
d.main(s);
```

This code is legal, but the compiler just resolves it back to the real class anyway ("OK, *d* is of type Duck, and main() is static, so I'll call the static main() in class Duck"). In other words, using *d* to invoke main() doesn't imply that main() will have any special knowledge of the object that *d* is referencing. It's just an alternate *way* to invoke a static method, but the method is still static!

Static variable:
value is the same for ALL
instances of the class

Imagine you wanted to count how many Duck instances are being created while your program is running. How would you do it? Maybe an instance variable that you increment in the constructor?

```
class Duck {
   int duckCount = 0;
    public Duck() {
       duckCount++;
    }
}
```

this would always set duckCount to 1 each time a Duck was made

No, that wouldn't work because duckCount is an instance variable, and starts at 0 for each Duck. You could try calling a method in some other class, but that's kludgey. You need a class that's got only a single copy of the variable, and all instances share that one copy.

That's what a static variable gives you: a value shared by all instances of a class. In other words, one value per *class*, instead of one value per *instance.*

The static duckCount variable is initialized ONLY when the class is first loaded, NOT each time a new instance is made.

```
public class Duck {

    private int size;
    private ███████ int duckCount = 0;

    public Duck() {
       duckCount++;
    }

    public void setSize(int s) {
       size = s;
    }
    public int getSize() {
       return size;
    }
}
```

Now it will keep incrementing each time the Duck constructor runs, because duckCount is static and won't be reset to 0.

A Duck object doesn't keep its own copy of duckCount.
Because duckCount is static, Duck objects all share a single copy of it. You can think of a static variable as a variable that lives in a CLASS instead of in an object

Each Duck object has its own size variable, but there's only one copy of the duckCount variable—the one in the class.

kid instance one

static variable: iceCream

kid instance two
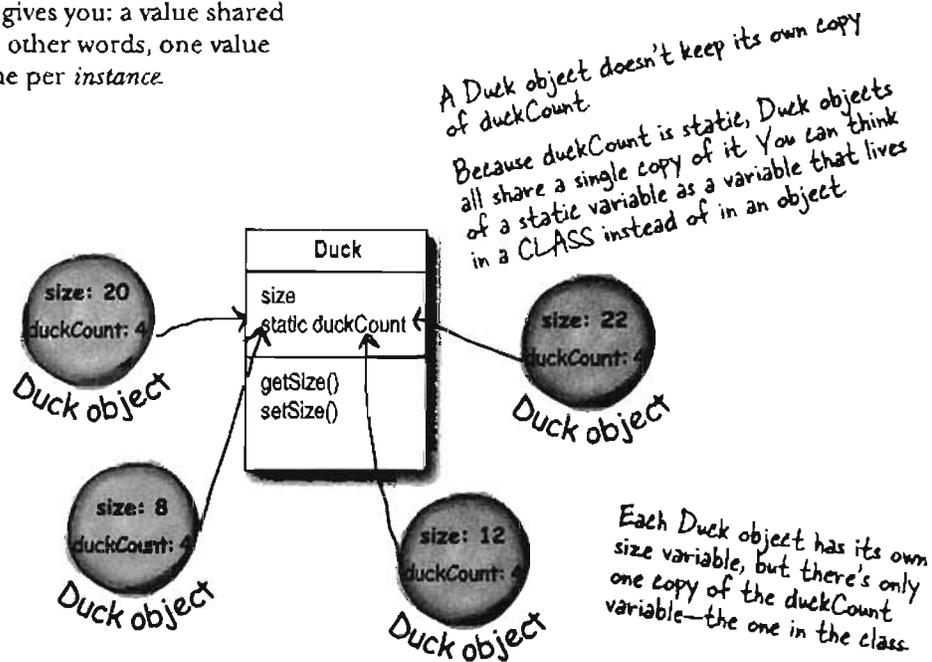
Static variables are shared.

All instances of the same class share a single copy of the static variables.

instance variables: 1 per **instance**

static variables: 1 per **class**

Brain Barbell

Earlier in this chapter, we saw that a private constructor means that the class can't be instantiated from code running outside the class. In other words, only code from within the class can make a new instance of a class with a private constructor. (There's a kind of chicken-and-egg problem here.)

What if you want to write a class in such a way that only ONE instance of it can be created, and anyone who wants to use an instance of the class will always use that one, single instance?

Initializing a static variable

Static variables are initialized when a *class is loaded.* A class is loaded because the JVM decides it's time to load it. Typically, the JVM loads a class because somebody's trying to make a new instance of the class, for the first time, or use a static method or variable of the class. As a programmer, you also have the option of telling the JVM to load a class, but you're not likely to need to do that. In nearly all cases, you're better off letting the JVM decide when to *load* the class.

And there are two guarantees about static initialization:

Static variables in a class are initialized before any *object* of that class can be created.

Static variables in a class are initialized before any *static method* of the class runs.

All static variables in a class are initialized *before* any object of that class can be created.

```
class Player {
    static int playerCount = 0;
    private String name;
    public Player(String n) {
        name = n;
        playerCount++;
    }
}

public class PlayerTestDrive {
    public static void main(String[] args) {
        System.out.println(Player.playerCount);
        Player one = new Player("Tiger Woods");
        System.out.println(Player.playerCount);
    }
}
```

The playerCount is initialized when the class is loaded. We explicitly initialized it to 0, but we don't need to since 0 is the default value for ints. Static variables get default values just like instance variables.

Default values for declared but uninitialized static and instance variables are the same:

Primitive integers (long, short, etc.): 0
Primitive floating points (float, double): 0.0
boolean: false

object references: null

Access a static variable just like a static method—with the class name.

Static variables are initialized when the class is loaded. If you don't explicitly initialize a static variable (by assigning it a value at the time you declare it), it gets a default value, so int variables are initialized to zero, which means we didn't need to explicitly say "playerCount = 0". Declaring, but not initializing, a static variable means the static variable will get the default value for that variable type, in exactly the same way that instance variables are given default values when declared.

```
File Edit Window Help What?

% java PlayerTestDrive
0 <— before any instances are made
1 <—
        after an object is created
```
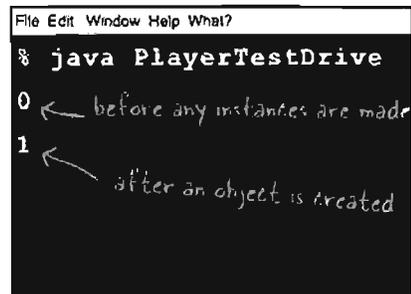
static <u>final</u> variables are constants

A variable marked **final** means that—once initialized—it can never change. In other words, the value of the static final variable will stay the same as long as the class is loaded. Look up Math.PI in the API, and you'll find:

public static final double PI = 3.141592653589793;

The variable is marked **public** so that any code can access it.

The variable is marked **static** so that you don't need an instance of class Math (which, remember, you're not allowed to create).

The variable is marked **final** because PI doesn't change (as far as Java is concerned).

There is no other way to designate a variable as a constant, but there is a naming convention that helps you to recognize one.
Constant variable names should be in all caps!

static initializer

Initialize a *final* static variable:

● **At the time you declare it:**

```
public class Foo {
    public static final int FOO_X = 25;
}
```

notice the naming convention — static final variables are constants, so the name should be all uppercase, with an underscore separating the words

OR

● **In a static initializer:**

```
public class Bar {
    public static final double BAR_SIGN;

    ▮▮▮▮ {
        BAR_SIGN = (double) Math.random();
    }
}
```

this code runs as soon as the class is loaded, before any static method is called and even before any static variable can be used.

If you don't give a value to a final variable in one of those two places:

```
public class Bar {
    public static final double BAR_SIGN;

}
```

no initialization!

The compiler will catch it:

```
File Edit Window Help Jack-In
% javac Bar.java

Bar.java:1: variable BAR_SIGN
might not have been initialized

1 error
```

final isn't just for static variables...

You can use the keyword **final** to modify non-static variables too, including instance variables, local variables, and even method parameters. In each case, it means the same thing: the value can't be changed. But you can also use final to stop someone from overriding a method or making a subclass.

non-static final **variables**

```
class Foof {
    final int size = 3;  ← now you can't change size
    final int whuffie;

    Foof() {
        whuffie = 42;  ← now you can't change whuffie
    }

    void doStuff(final int x) {
        // you can't change x
    }

    void doMore() {
        final int z = 7;
        // you can't change z
    }
}
```

final method

```
class Poof {
    final void calcWhuffie() {
        // important things
        // that must never be overridden
    }
}
```

final class

```
final class MyMostPerfectClass {
    // cannot be extended
}
```

A final variable means you can't change its value.

A final method means you can't override the method.

A final class means you can't extend the class (i.e. you can't make a subclass).

> It's all so... so *final*. I mean, if I'd *known* I wouldn't be able to change things...

there are no
Dumb Questions

Q: A static method can't access a non-static variable. But can a non-static method access a static variable?

A: Of course. A non-static method in a class can always call a static method in the class or access a static variable of the class.

Q: Why would I want to make a class final? Doesn't that defeat the whole purpose of OO?

A: Yes and no. A typical reason for making a class final is for security. You can't, for example, make a subclass of the String class. Imagine the havoc if someone extended the String class and substituted their own String subclass objects, polymorphically, where String objects are expected. If you need to count on a particular implementation of the methods in a class, make the class final.

Q: Isn't it redundant to have to mark the methods final if the class is final?

A: If the class is final, you don't need to mark the methods final. Think about it—if a class is final it can never be subclassed, so none of the methods can ever be overridden.

On the other hand, if you *do* want to allow others to extend your class, and you want them to be able to override some, but not all, of the methods, then don't mark the class final but go in and selectively mark specific methods as final. A final method means that a subclass can't override that particular method.

BULLET POINTS

- A *static method* should be called using the class name rather than an object reference variable: `Math.random()` vs. `myFoo.go()`

- A static method can be invoked without any instances of the method's class on the heap.

- A static method is good for a utility method that does not (and will never) depend on a particular instance variable value.

- A static method is not associated with a particular instance—only the class—so it cannot access any instance variable values of its class. It wouldn't know *which* instance's values to use.

- A static method cannot access a non-static method, since non-static methods are usually associated with instance variable state.

- If you have a class with only static methods, and you do not want the class to be instantiated, you can mark the constructor private.

- A *static variable* is a variable shared by all members of a given class. There is only one copy of a static variable in a class, rather than one copy per each individual instance for instance variables.

- A static method can access a static variable.

- To make a constant in Java, mark a variable as both static and final.

- A final static variable must be assigned a value either at the time it is declared, or in a static initializer.
```
static {
    DOG_CODE = 420;
}
```

- The naming convention for constants (final static variables) is to make the name all uppercase.

- A final variable value cannot be changed once it has been assigned.

- Assigning a value to a final *instance* variable must be either at the time it is declared, or in the constructor.

- A final method cannot be overridden.

- A final class cannot be extended (subclassed).

Sharpen your pencil

What's Legal?

Given everything you've just learned about static and final, which of these would compile?

KEEP ← RIGHT

```
public class Foo {
    static int x;

    public void go() {
        System.out.println(x);
    }
}
```

```
public class Foo4 {
    static final int x = 12;

    public void go() {
        System.out.println(x);
    }
}
```

```
public class Foo2 {
    int x;

    public static void go() {
        System.out.println(x);
    }
}
```

```
public class Foo5 {
    static final int x = 12;

    public void go(final int x) {
        System.out.println(x);
    }
}
```

```
public class Foo3 {
    final int x;

    public void go() {
        System.out.println(x);
    }
}
```

```
public class Foo6 {
    int x = 12;

    public static void go(final int x) {
        System.out.println(x);
    }
}
```

Math methods

Now that we know how static methods work, let's look at some static methods in class Math. This isn't all of them, just the highlights. Check your API for the rest including sqrt(), tan(), ceil(), floor(), and asin().

Math.random()

Returns a double between 0.0 through (but not including) 1.0.

```
double r1 = Math.random();
int r2 = (int) (Math.random() * 5);
```

Math.abs()

Returns a double that is the absolute value of the argument. The method is overloaded, so if you pass it an int it returns an int. Pass it a double it returns a double.

```
int x = Math.abs(-240);   // returns 240
double d = Math.abs(240.45);  // returns  240.45
```

Math.round()

Returns an int or a long (depending on whether the argument is a float or a double) rounded to the nearest integer value.

```
int x = Math.round(-24.8f);  // returns -25
int y = Math.round(24.45f);  // returns 24
```

Remember, floating point literals are assumed to be doubles unless you add the 'f'.

Math.min()

Returns a value that is the minimum of the two arguments. The method is overloaded to take ints, longs, floats, or doubles.

```
int x = Math.min(24,240);  // returns 24
double y = Math.min(90876.5, 90876.49);  // returns 90876.49
```

Math.max()

Returns a value that is the maximum of the two arguments. The method is overloaded to take ints, longs, floats, or doubles.

```
int x = Math.max(24,240);  // returns 240
double y = Math.max(90876.5, 90876.49);  // returns 90876.5
```

Wrapping a primitive

Sometimes you want to treat a primitive like an object. For example, in all versions of Java prior to 5.0, you cannot put a primitive directly into a collection like ArrayList or HashMap:

```
int x = 32;
ArrayList list = new ArrayList();
list.add(x);
```

This won't work unless you're using Java 5.0 or greater!! There's no add(int) method in ArrayList that takes an int! (ArrayList only has add() methods that take object references, not primitives.)

There's a wrapper class for every primitive type, and since the wrapper classes are in the java.lang package, you don't need to import them. You can recognize wrapper classes because each one is named after the primitive type it wraps, but with the first letter capitalized to follow the class naming convention.

Oh yeah, for reasons absolutely nobody on the planet is certain of, the API designers decided not to map the names *exactly* from primitive type to class type. You'll see what we mean:

Boolean

Character

Byte

Short

Integer

Long

Float

Double

Watch out! The names aren't mapped exactly to the primitive types. The class names are fully spelled out.

wrapping a value

Give the primitive to the wrapper constructor. That's it.

```
int i = 288;
Integer iWrap = new Integer(i);
```

All the wrappers work like this. Boolean has a booleanValue(), Character has a charValue(), etc.

_un_wrapping a value

```
int unWrapped = iWrap.intValue();
```
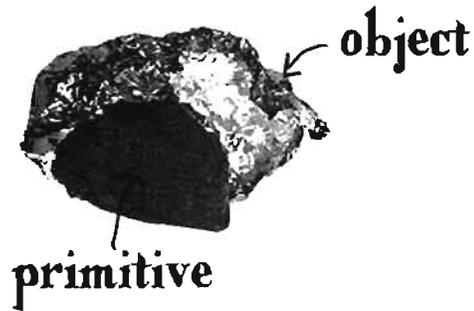
object

primitive

When you need to treat a primitive like an object, wrap it. If you're using any version of Java before 5.0, you'll do this when you need to store a primitive value inside a collection like ArrayList or HashMap.

int primitive

Integer object

Integer object
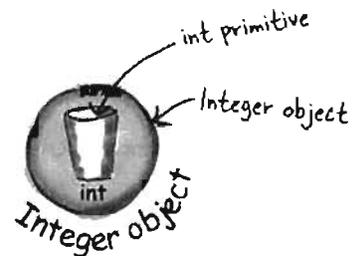
Note: the picture at the top is a chocolate in a foil wrapper. Get it? Wrapper? Some people think it looks like a baked potato, but that works too.

> This is stupid. You mean I can't just make an ArrayList of ints??? I have to wrap every single frickin' one in a new Integer object, then unwrap it when I try to access that value in the ArrayList? That's a waste of time and an error waiting to happen...

Before Java 5.0, YOU had to do the work...

She's right. In all versions of Java prior to 5.0, primitives were primitives and object references were object references, and they were NEVER treated interchangeably. It was always up to you, the programmer, to do the wrapping and unwrapping. There was no way to pass a primitive to a method expecting an object reference, and no way to assign the result of a method returning an object reference directly to a primitive variable—even when the returned reference is to an Integer and the primitive variable is an int. There was simply no relationship between an Integer and an int, other than the fact that Integer has an instance variable of type int (to hold the primitive the Integer wraps). All the work was up to you.

An ArrayList of primitive ints

Without autoboxing (Java versions *before* 5.0)

```
public void doNumsOldWay() {

    ArrayList listOfNumbers = new ArrayList();

    listOfNumbers.add(new Integer(3));

    Integer one = (Integer) listOfNumbers.get(0);

    int intOne = one.intValue();
}
```

Make an ArrayList (Remember, before 5.0 you could not specify the TYPE, so all ArrayLists were lists of Objects.)

You can't add the primitive '3' to the list, so you have to wrap it in an Integer first.

It comes out as type Object, but you can cast the Object to an Integer.

Finally you can get the primitive out of the Integer.

Autoboxing: blurring the line between primitive and object

The autoboxing feature added to Java 5.0 does the conversion from primitive to wrapper object *automatically!*

Let's see what happens when we want to make an ArrayList to hold ints.

An ArrayList of primitive ints

With autoboxing (Java versions 5.0 or greater)

```
public void doNumsNewWay() {

    ArrayList<Integer> listOfNumbers = new ArrayList<Integer>();

    listOfNumbers.add(3);    Just add it!

    int num = listOfNumbers.get(0);
}
```

Make an ArrayList of type Integer.

And the compiler automatically unwraps (unboxes) the Integer object so you can assign the int value directly to a primitive without having to call the intValue() method on the Integer object.

Although there is NOT a method in ArrayList for add(int), the compiler does all the wrapping (boxing) for you. In other words, there really IS an Integer object stored in the ArrayList, but you get to "pretend" that the ArrayList takes ints. (You can add both ints and Integers to an ArrayList<Integer>.)

Q: Why not declare an ArrayList<int> if you want to hold ints?

A: Because... *you can't.* Remember, the rule for generic types is that you can specify only class or interface types, *not primitives.* So ArrayList<int> will not compile. But as you can see from the code above, it doesn't really matter, since the compiler lets you put ints into the ArrayList<Integer>. In fact, there's really no way to *prevent* you from putting primitives into an ArrayList where the type of the list is the type of that primitive's wrapper, if you're using a Java 5.0-compliant compiler, since autoboxing will happen automatically. So, you can put boolean primitives in an ArrayList<Boolean> and chars into an ArrayList<Character>.

Autoboxing works almost everywhere

Autoboxing lets you do more than just the obvious wrapping and unwrapping to use primitives in a collection... it also lets you use either a primitive or its wrapper type virtually anywhere one or the other is expected. Think about that!

Fun with autoboxing

Method arguments

If a method takes a wrapper type, you can pass a reference to a wrapper or a primitive of the matching type. And of course the reverse is true—if a method takes a primitive, you can pass in either a compatible primitive or a reference to a wrapper of that primitive type.
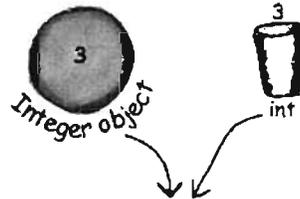

```
void takeNumber(Integer i) { }
```


Return values

If a method declares a primitive return type, you can return either a compatible primitive or a reference to the wrapper of that primitive type. And if a method declares a wrapper return type, you can return either a reference to the wrapper type or a primitive of the matching type.

```
int giveNumber() {
    return x;
}
```
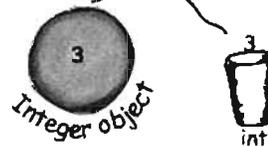
Boolean expressions

Any place a boolean value is expected, you can use either an expression that evaluates to a boolean (4 > 2), or a primitive boolean, or a reference to a Boolean wrapper.
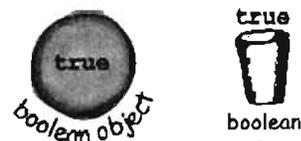

```
if (bool) {
    System.out.println("true");
}
```

---

## Operations on numbers

This is probably the strangest one—yes, you can now use a wrapper type as an operand in operations where the primitive type is expected. That means you can apply, say, the increment operator against a reference to an Integer object!
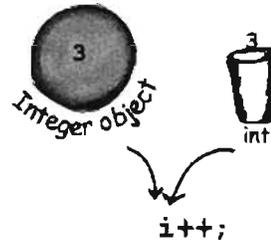
But don't worry—this is just a compiler trick. The language wasn't modified to make the operators work on objects; the compiler simply converts the object to its primitive type before the operation. It sure looks weird, though.

```
Integer i = new Integer(42);
i++;
```

And that means you can also do things like:

```
Integer j = new Integer(5);
Integer k = j + 3;
```


`i++;`

Assignments

You can assign either a wrapper or primitive to a variable declared as a matching wrapper or primitive. For example, a primitive int variable can be assigned to an Integer reference variable, and vice-versa—a reference to an Integer object can be assigned to a variable declared as an int primitive.

`Double d = x;`
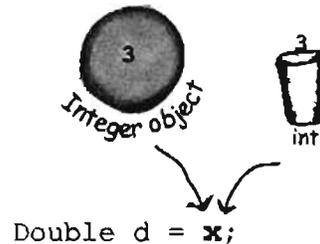
Sharpen your pencil

Will this code compile? Will it run? If it runs, what will it do?

Take your time and think about this one; it brings up an implication of autoboxing that we didn't talk about.

You'll have to go to your compiler to find the answers. (Yes, we're forcing you to experiment, for your own good of course.)

```
public class TestBox {

    Integer i;
    int j;

    public static void main (String[] args) {
        TestBox t = new TestBox();
        t.go();
    }

    public void go() {
        j=i;
        System.out.println(j);
        System.out.println(i);
    }
}
```

But wait! There's more! Wrappers have static utility methods too!

Besides acting like a normal class, the wrappers have a bunch of really useful static methods. We've used one in this book before—Integer.parseInt().

The parse methods take a String and give you back a primitive value.

Converting a String to a primitive value is easy:

No problem to parse "2" into 2

```
String s = "2";
int x = Integer.parseInt(s);
double d = Double.parseDouble("420.24");

boolean b = new Boolean("true").booleanValue();
```

You'd think there would be a Boolean.parseBoolean() wouldn't you? But there isn't. Fortunately there's a Boolean constructor that takes (and parses) a String, and then you just get the primitive value by unwrapping it.

But if you try to do this:

```
String t = "two";
int y = Integer.parseInt(t);
```

Uh-oh. This compiles just fine, but at runtime it blows up. Anything that can't be parsed as a number will cause a NumberFormatException

You'll get a runtime exception:

```
File Edit Window Help Clue
% java Wrappers
Exception in thread "main"
java.lang.NumberFormatException: two
at java.lang.Integer.parseInt(Integer.java:409)
at java.lang.Integer.parseInt(Integer.java:458)
at Wrappers.main(Wrappers.java:9)
```

Every method or constructor that parses a String can throw a NumberFormatException. It's a runtime exception, so you don't have to handle or declare it. But you might want to.

(We'll talk about Exceptions in the next chapter.)

And now in reverse... turning a primitive number into a String

There are several ways to turn a number into a String. The easiest is to simply concatenate the number to an existing String.

```
double d = 42.5;
String doubleString = "" + d;
```

Remember the '+' operator is overloaded in Java (the only overloaded operator) as a String concatenator. Anything added to a String becomes Stringified.

```
double d = 42.5;
String doubleString = Double.toString(d);
```

Another way to do it using a static method in class Double.

Yeah, but how do I make it look like money? With a dollar sign and two decimal places like $56.87 or what if I want commas like 45,687,890 or what if I want it in...

Where's my printf like I have in C? Is number formatting part of the I/O classes?

Number formatting

In Java, formatting numbers and dates doesn't have to be coupled with I/O. Think about it. One of the most typical ways to display numbers to a user is through a GUI. You put Strings into a scrolling text area, or maybe a table. If formatting was built only into print statements, you'd never be able to format a number into a nice String to display in a GUI. Before Java 5.0, most formatting was handled through classes in the java.text package that we won't even look at in this version of the book, now that things have changed.

In Java 5.0, the Java team added more powerful and flexible formatting through a Formatter class in java.util. But you don't need to create and call methods on the Formatter class yourself, because Java 5.0 added convenience methods to some of the I/O classes (including printf()) and the String class. So it's a simple matter of calling a static String.format() method and passing it the thing you want formatted along with formatting instructions.

Of course, you do have to know how to supply the formatting instructions, and that takes a little effort unless you're familiar with the *printf()* function in C/C++. Fortunately, even if you *don't* know printf() you can simply follow recipes for the most basic things (that we're showing in this chapter). But you *will* want to learn how to format if you want to mix and match to get *anything* you want.

We'll start here with a basic example, then look at how it works. (Note: we'll revisit formatting again in the I/O chapter.)

Formatting a number to use commas

```
public class TestFormats {

    public static void main (String[] args) {

        String s = String.format("%, d", 1000000000);
        System.out.println(s);
    }
}
```

The number to format (we want it to have commas).

The formatting instructions for how to format the second argument (which in this case is an int value). Remember, there are only two arguments to this method here—the first comma is INSIDE the String literal, so it isn't separating arguments to the format method.

```
1,000,000,000
```

Now we get commas inserted into the number.

Formatting deconstructed...

At the most basic level, formatting consists of two main parts (there is more, but we'll start with this to keep it cleaner):

① Formatting Instructions

You use special format specifiers that describe how the argument should be formatted.

② The argument to be formatted.

Although there can be more than one argument, we'll start with just one. The argument type can't be just *anything...* it has to be something that can be formatted using the format specifiers in the formatting instructions. For example, if your formatting instructions specify a *floating point number*, you can't pass in a Dog or even a String that looks like a floating point number.

Note: if you already know printf() from c/C++, you can probably just skim the next few pages. Otherwise, read carefully!

Do this...   to this.

```
format("%, d", 1000000000);
```

Use these instructions... on this argument.

What do these instructions actually say?

"Take the second argument to this method, and format it as a **decimal integer and insert commas.**"

How do they say that?

On the next page we'll look in more detail at what the syntax "%, d" actually means, but for starters, any time you see the percent sign (%) in a format String (which is always the first argument to a format() method), think of it as representing a variable, and the variable is the other argument to the method. The rest of the characters after the percent sign describe the formatting instructions for the argument.

The percent (%) says, "insert argument here" (and format it using these instructions)

The first argument to a format() method is called the format String, and it can actually include characters that you just want printed as-is, without extra formatting. When you see the % sign, though, think of the percent sign as a variable that represents the other argument to the method.

Characters to include in the final String returned from format().

Format specifiers for the second argument to the method (the number).

More characters to include in the String after the second argument is formatted and inserted.

Argument to be formatted.

```
format("I have %.2f bugs to fix.", 476578.09876);
```

Output

```
I have 476578.10 bugs to fix.
```

Notice we lost some of the numbers after the decimal point. Can you guess what the ".2f" means?

The "%" sign tells the formatter to insert the other method argument (the second argument to format(), the number) here, AND format it using the ".2f" characters after the percent sign. Then the rest of the format String, "bugs to fix", is added to the final output.

Adding a comma

```
format("I have %,.2f bugs to fix.", 476578.09876);
```

```
I have 476,578.10 bugs to fix.
```

By changing the format instructions from "%.2f" to %,.2f", we got a comma in the formatted number.

But how does it even KNOW where the instructions end and the rest of the characters begin? How come it doesn't print out the "f" in "%.2f"? Or the "2"? How does it know that the .2f was part of the instructions and NOT part of the String?

The format String uses its own little language syntax

You obviously can't put just *anything* after the "%" sign. The syntax for what goes after the percent sign follows very specific rules, and describes how to format the argument that gets inserted at that point in the result (formatted) String.

You've already seen two examples:

%, d means "insert commas and format the number as a decimal integer."

and

%.2f means "format the number as a floating point with a precision of two decimal places."

and

%,.2f means "insert commas and format the number as a floating point with a precision of two decimal places."

The real question is really, "How do I know what to put after the percent sign to get it to do what I want?" And that includes knowing the symbols (like "d" for decimal and "f" for floating point) as well as the order in which the instructions must be placed following the percent sign. For example, if you put the comma after the "d" like this: "%d," instead of "%,d" it won't work!

Or will it? What do you think this will do:

```
String.format("I  have %.2f, bugs to fix.", 476578.09876);
```

(We'll answer that on the next page.)

The format specifier

Everything after the percent sign up to and including the type indicator (like "d" or "f") are part of the formatting instructions. After the type indicator, the formatter assumes the next set of characters are meant to be part of the output String, until or unless it hits another percent (%) sign. Hmmmm... is that even possible? Can you have more than one formatted argument variable? Put that thought on hold for right now; we'll come back to it in a few minutes. For now, let's look at the syntax for the format specifiers—the things that go after the percent (%) sign and describe how the argument should be formatted.

A format specifier can have up to five different parts (not including the "%"). Everything in brackets [ ] below is optional, so only the percent (%) and the type are required. But the order is also mandatory, so any parts you DO use must go in this order.

`%[argument number][flags][width][.precision]`**type**

We'll get to this later... it lets you say WHICH argument if there's more than one. (Don't worry about it just yet.)

These are for special formatting options like inserting commas, or putting negative numbers in parentheses, or to make the numbers left justified.

This defines the MINIMUM number of characters that will be used. That's *minimum* not TOTAL. If the number is longer than the width, it'll still be used in full, but if it's less than the width, it'll be padded with zeroes.

You already know this one...it defines the precision. In other words, it sets the number of decimal places. Don't forget to include the "." in there.

Type is mandatory (see the next page) and will usually be "d" for a decimal integer or "f" for a floating point number.

`%[argument number][flags][width][.precision]`**type**

`format("%,6.1f", 42.000);`

There's no "argument number" specified in this format String, but all the other pieces are there.

The only required specifier is for TYPE

Although type is the only required specifier, remember that if you *do* put in anything else, type must always come last! There are more than a dozen different type modifiers (not including dates and times; they have their own set), but most of the time you'll probably use %d (decimal) or %f (floating point). And typically you'll combine %f with a precision indicator to set the number of decimal places you want in your output.

The TYPE is mandatory, everything else is optional.

%d   ***decimal***

```
format("%d", 42);
```

> A 42.25 would not work! It would be the same as trying to directly assign a double to an int variable.

```
42
```

The argument must be compatible with an int, so that means only byte, short, int, and char (or their wrapper types).

%f   ***floating point***

```
format("%.3f", 42.000000);
```

> Here we combined the "f" with a precision indicator ".3" so we ended up with three zeroes.

```
42.000
```

The argument must be of a floating point type, so that means only a float or double (primitive or wrapper) as well as something called BigDecimal (which we don't look at in this book).

%x   ***hexadecimal***

```
format("%x", 42);
```

```
2a
```

The argument must be a byte, short, int, long (including both primitive and wrapper types), and BigInteger.

%c   ***character***

```
format("%c", 42);
```

> The number 42 represents the char "*".

```
*
```

The argument must be a byte, short, char, or int (including both primitive and wrapper types).

You must include a type in your format instructions, and if you specify things besides type, the type must always come <u>last</u>. Most of the time, you'll probably format numbers using either "d" for decimal or "f" for floating point.

What happens if I have more than one argument?

Imagine you want a String that looks like this:

"The rank is *20,456,654* out of *100,567,890.24*."

But the numbers are coming from variables. What do you do? You simply add *two* arguments after the format String (first argument), so that means your call to format() will have three arguments instead of two. And inside that first argument (the format String), you'll have two different format specifiers (two things that start with "%"). The first format specifier will insert the second argument to the method, and the second format specifier will insert the third argument to the method. In other words, the variable insertions in the format String use the order in which the other arguments are passed into the format() method.

```
int one = 20456654;
double two = 100567890.248907;
String s = String.format("The rank is %,d out of %,.2f", one, two);
```
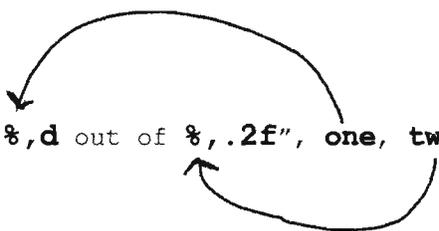
```
The rank is 20,456,654 out of 100,567,890.25
```

We added commas to both variables, and restricted the floating point number (the second variable) to two decimal places.

When you have more than one argument, they're inserted using the order in which you pass them to the format() method.

As you'll see when we get to date formatting, you might actually want to apply different formatting specifiers to the same argument. That's probably hard to imagine until you see how *date* formatting (as opposed to the *number* formating we've been doing) works. Just know that in a minute, you'll see how to be more specific about which format specifiers are applied to which arguments.

Q: Um, there's something REALLY strange going on here. Just how many arguments *can* I pass? I mean, how many overloaded format() methods are IN the String class? So, what happens if I want to pass, say, ten different arguments to be formatted for a single output String?

A: Good catch. Yes, there *is* something strange (or at least new and different) going on, and no there are *not* a bunch of overloaded format() methods to take a different number of possible arguments. In order to support this new formatting (printf-like) API in Java, the language needed another new feature—*variable argument lists* (called *varargs* for short). We'll talk about varargs only in the appendix because outside of formatting, you probably won't use them much in a well-designed system.

So much for numbers, what about dates?

Imagine you want a String that looks like this: "Sunday, Nov 28 2004"

Nothing special there, you say? Well, imagine that all you have to start with is a variable of type Date—A Java class that can represent a timestamp, and now you want to take that object (as opposed to a number) and send it through the formatter.

The main difference between number and date formatting is that date formats use a two-character type that starts with "t" (as opposed to the single character "f" or "d", for example). The examples below should give you a good idea of how it works:

The complete date and time   %tc

```
String.format("%tc", new Date());
```

```
Sun Nov 28 14:52:41 MST 2004
```

Just the time   %tr

```
String.format("%tr", new Date());
```

```
03:01:47 PM
```

Day of the week, month and day   %tA %tB %td

There isn't a single format specifier that will do exactly what we want, so we have to combine three of them for day of the week (%tA), month (%tB), and day of the month (%td).

```
Date today = new Date();
String.format("%tA, %tB %td",today,today,today)
```

The comma is not part of the formatting... it's just the character we want printed after the first inserted formatted argument.

But that means we have to pass the Date object in three times, one for each part of the format that we want. In other words, the %tA will give us just the day of the week, but then we have to do it again to get just the month and again for the day of the month.

```
Sunday, November 28
```

Same as above, but *without duplicating the arguments*   %tA %tB %td

```
Date today = new Date();
String.format("%tA, %<tB %<td",today);
```

You can think of this as kind of like calling three different getter methods on the Date object, to get three different pieces of data from it.

The angle-bracket "<" is just another flag in the specifier that tells the formatter to "use the previous argument again." So it saves you from repeating the arguments, and instead you format the same argument three different ways.
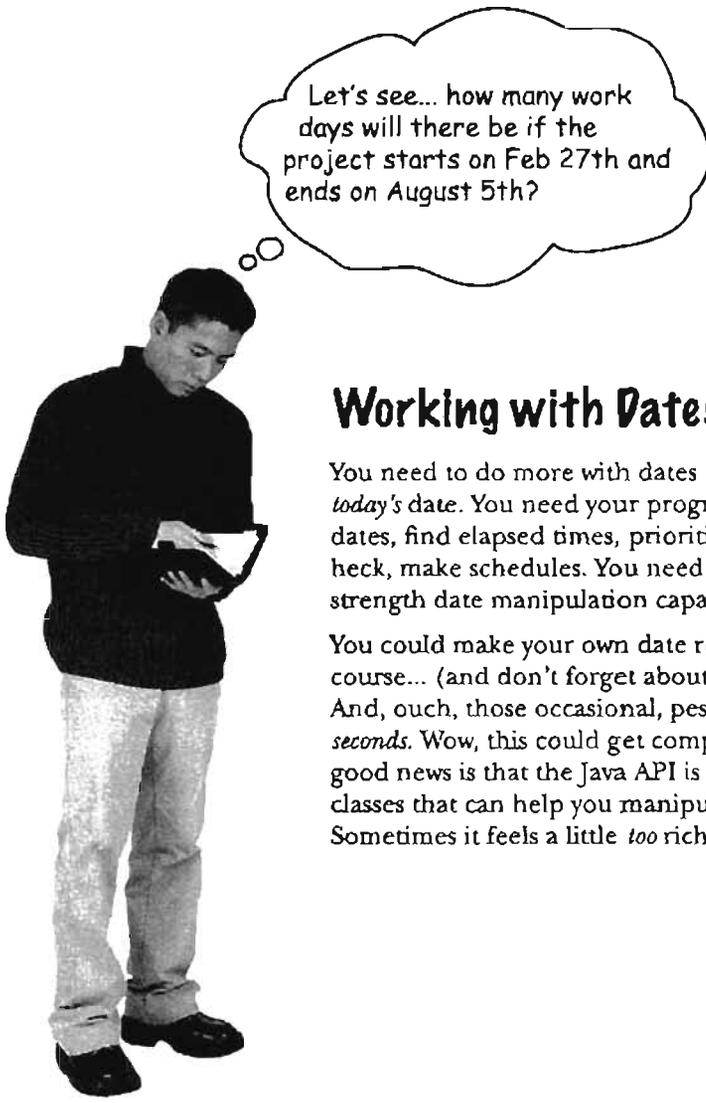
Let's see... how many work days will there be if the project starts on Feb 27th and ends on August 5th?

Working with Dates

You need to do more with dates than just get *today's* date. You need your programs to adjust dates, find elapsed times, prioritize schedules, heck, make schedules. You need industrial strength date manipulation capabilities.

You could make your own date routines of course... (and don't forget about leap years!) And, ouch, those occasional, pesky leap-*seconds*. Wow, this could get complicated. The good news is that the Java API is rich with classes that can help you manipulate dates. Sometimes it feels a little *too* rich...

Moving backward and forward in time

Let's say your company's work schedule is Monday through Friday. You've been assigned the task of figuring out the last work day in each calendar month this year...

It seems that java.util.Date is actually... out of date

Earlier we used java.util.Date to find today's date, so it seems logical that this class would be a good place to start looking for some handy date manipulation capabilities, but when you check out the API you'll find that most of Date's methods have been deprecated!

The Date class is still great for getting a "time stamp"—an object that represents the current date and time, so use it when you want to say, "give me NOW".

The good news is that the API recommends **java.util.Calendar** instead, so let's take a look:

Use java.util.Calendar for your date *manipulation*

The designers of the Calendar API wanted to think globally, literally. The basic idea is that when you want to work with dates, you ask for a Calendar (through a static method of the Calendar class that you'll see on the next page), and the JVM hands you back an instance of a concrete subclass of Calendar. (Calendar is actually an abstract class, so you're always working with a concrete subclass.)

More interesting, though, is that the *kind* of calendar you get back will be *appropriate for your locale*. Much of the world uses the Gregorian calendar, but if you're in an area that doesn't use a Gregorian calendar you can get Java libraries to handle other calendars such as Buddhist, or Islamic or Japanese.

The standard Java API ships with **java.util.GregorianCalendar**, so that's what we'll be using here. For the most part, though, you don't even have to think about the kind of Calendar subclass you're using, and instead focus only on the methods of the Calendar class.

For a time-stamp of "now", use Date. But for everything else, use Calendar.

Getting an object that extends Calendar

How in the world do you get an "instance" of an abstract class?
Well you don't of course, this won't work:

This WON'T work:

```
Calendar cal = new Calendar();
```

The compiler won't allow this!

Instead, use the static "getInstance()" method:

```
Calendar cal = Calendar.getInstance();
```

This syntax should look familiar at this point — we're invoking a static method.

> Wait a minute.
> If you can't make an instance of the Calendar class, what exactly are you assigning to that Calendar reference?

You can't get an instance of Calendar, but you can can get an instance of a concrete Calendar subclass.

Obviously you can't get an instance of Calendar, because Calendar is abstract. But you're still free to call static methods on Calendar, since *static* methods are called on the *class*, rather than on a particular instance. So you call the static getInstance() on Calendar and it gives you back... an instance of a concrete subclass. Something that extends Calendar (which means it can be polymorphically assigned to Calendar) and which—by contract—can respond to the methods of class Calendar.

In most of the world, and by default for most versions of Java, you'll be getting back a **java.util.GregorianCalendar** instance.

Working with Calendar objects

There are several key concepts you'll need to understand in order to work with Calendar objects:

- **Fields hold** *state* - A Calendar object has many fields that are used to represent aspects of its ultimate state, its date and time. For instance, you can get and set a Calendar's *year* or *month*.

- **Dates and Times can be** *incremented* - The Calendar class has methods that allow you to add and subtract values from various fields, for example "add one to the month", or "subtract three years".

- **Dates and Times can be represented in** *milliseconds* - The Calendar class lets you convert your dates into and out of a millisecond representation. (Specifically, the number of milliseconds that have occured since January 1st, 1970.) This allows you to perform precise calculations such as "elapsed time between two times" or "add 63 hours and 23 minutes and 12 seconds to this time".

An example of working with a Calendar object:

```
Calendar c = Calendar.getInstance();

c.set(2004,0,7,15,40);

long day1 = c.getTimeInMillis();

day1 += 1000 * 60 * 60;

c.setTimeInMillis(day1);

System.out.println("new hour " + c.get(c.HOUR_OF_DAY));

c.add(c.DATE, 35);

System.out.println("add 35 days " + c.getTime());

c.roll(c.DATE, 35);

System.out.println("roll 35 days " + c.getTime());

c.set(c.DATE, 1);

System.out.println("set to 1 " + c.getTime());
```

Set time to Jan. 7, 2004 at 15:40. (Notice the month is zero-based.)

Convert this to a big ol' amount of milliseconds.

Add an hour's worth of millis, then update the time. (Notice the "+=", it's like day1 = day1 + ...).

Add 35 days to the date, which should move us into February.

"Roll" 35 days onto this date. This "rolls" the date ahead 35 days, but DOES NOT change the month!

We're not incrementing here, just doing a "set" of the date.

```
File Edit Window Help Time-Flies
new hour 16
add 35 days Wed Feb 11 16:40:41 MST 2004
roll 35 days Tue Feb 17 16:40:41 MST 2004
set to 1 Sun Feb 01 16:40:41 MST 2004
```

This output confirms how millis, add, roll, and set work.

Highlights of the Calendar API

We just worked through using a few of the fields and methods in the Calendar class. This is a big API, so we're showing only a few of the most common fields and methods that you'll use. Once you get a few of these it should be pretty easy to bend the rest of the this API to your will.

Key Calendar Methods

add(int field, int amount)
> Adds or subtracts time from the calendar's field.

get(int field)
> Returns the value of the given calendar field.

getInstance()
> Returns a Calendar, you can specify a locale.

getTimeInMillis()
> Returns this Calendar's time in millis, as a long.

roll(int field, boolean up)
> Adds or subtracts time without changing larger fields.

set(int field, int value)
> Sets the value of a given Calendar field.

set(year, month, day, hour, minute) (all ints)
> A common variety of set to set a complete time.

setTimeInMillis(long millis)
> Sets a Calendar's time based on a long milli-time.

// more...

Key Calendar Fields

DATE / DAY_OF_MONTH
> Get / set the day of month

HOUR / HOUR_OF_DAY
> Get / set the 12 hour or 24 hour value.

MILLISECOND
> Get / set the milliseconds.

MINUTE
> Get / set the minute.

MONTH
> Get / set the month.

YEAR
> Get / set the year.

ZONE_OFFSET
> Get / set raw offset of GMT in millis.

// more...

Even more Statics!... static imports

New to Java 5.0... a real mixed blessing. Some people love this idea, some people hate it. Static imports exist only to save you some typing. If you hate to type, you might just like this feature. The downside to static imports is that - if you're not careful - using them can make your code a lot harder to read.

The basic idea is that whenever you're using a static class, a static variable, or an enum (more on those later), you can import them, and save yourself some typing.

Use Carefully:

static imports can make your code confusing to read

Some old-fashioned code:

```
import java.lang.Math;

class NoStatic {

   public static void main(String [] args) {

      System.out.println("sqrt " + Math.sqrt(2.0));

      System.out.println("tan " + Math.tan(60));

   }

}
```

Same code, with static imports:

The syntax to use when declaring static imports.

```
import static java.lang.System.out;

import static java.lang.Math.*;

class WithStatic {

   public static void main(String [] args) {

      out.println("sqrt " + sqrt(2.0));

      out.println("tan " + tan(60));

   }

}
```

Static imports in action.

- Caveats & Gotchas

- If you're only going to use a static member a few times, we think you should avoid static imports, to help keep the code more readable.

- If you're going to use a static member a lot, (like doing lots of Math calculations), then it's probably OK to use the static import.

- Notice that you can use wildcards (.*), in your static import declaration.

- A big issue with static imports is that it's not too hard to create naming conflicts. For example, if you have two different classes with an "add()" method, how will you and the compiler know which one to use?

Fireside Chats

Tonight's Talk: An instance variable takes cheap shots at a static variable

Instance Variable

I don't even know why we're doing this. Everyone knows static variables are just used for constants. And how many of those are there? I think the whole API must have, what, four? And it's not like anybody ever uses them.

Static Variable

You really should check your facts. When was the last time you looked at the API? It's frickin' loaded with statics! It even has entire classes dedicated to holding constant values. There's a class called SwingConstants, for example, that's just full of them.

Full of it. Yeah, you can say that again. OK, so there are a few in the Swing library, but everybody knows Swing is just a special case.

It might be a special case, but it's a really important one! And what about the Color class? What a pain if you had to remember the RGB values to make the standard colors? But the color class already has constants defined for blue, purple, white, red, etc. Very handy.

Ok, but besides a few GUI things, give me an example of just one static variable that anyone would actually use. In the real world.

How's System.out for starters? The out in System.out is a static variable of the System class. You personally don't make a new instance of the System, you just ask the System class for its out variable.

Well, that's another special case. And nobody uses that except for debugging anyway.

Oh, like debugging isn't important?

And here's something that probably never crossed your narrow mind—let's face it, static variables are more efficient. One per class instead of one per instance. The memory savings might be huge!

Instance Variable

Um, aren't you forgetting something?

Static variables are about as un-OO as it gets!! Gee why not just go take a giant backwards step and do some procedural programming while we're at it.

You're like a global variable, and any programmer worth his PDA knows that's usually a Bad Thing.

Yeah you live in a class, but they don't call it *Class*-Oriented programming. That's just stupid. You're a relic. Something to help the old-timers make the leap to java.

Well, OK, every once in a while sure, it makes sense to use a static, but let me tell you, abuse of static variables (and methods) is the mark of an immature OO programmer. A designer should be thinking about *object* state, not *class* state.

Static methods are the worst things of all, because it usually means the programmer is thinking procedurally instead of about objects doing things based on their unique object state.

Riiiiiight. Whatever you need to tell yourself...

Static Variable

What?

What do you mean *un*-OO?

I am NOT a global variable. There's no such thing. I live in a class! That's pretty OO you know, a CLASS. I'm not just sitting out there in space somewhere; I'm a natural part of the state of an object; the only difference is that I'm shared by all instances of a class. Very efficient.

Alright just stop right there. THAT is definitely not true. Some static variables are absolutely crucial to a system. And even the ones that aren't crucial sure are handy.

Why do you say that? And what's wrong with static methods?

Sure, I know that objects should be the focus of an OO design, but just because there are some clueless programmers out there... don't throw the baby out with the bytecode. There's a time and place for statics, and when you need one, nothing else beats it.

Exercise

BE the compiler

The Java file on this page represents a complete program. Your job is to play compiler and determine whether this file will compile. If it won't compile, how would you fix it, and if it does compile, what would be its output?

```java
class StaticSuper{

   static {
      System.out.println("super static block");
   }

   StaticSuper{
      System.out.println(
         "super constructor");
   }
}

public class StaticTests extends StaticSuper {
   static int rand;

   static {
      rand = (int) (Math.random() * 6);
      System.out.println("static block " + rand);
   }

   StaticTests() {
      System.out.println("constructor");
   }

   public static void main(String [] args) {
      System.out.println("in main");
      StaticTests st = new StaticTests();
   }
}
```

If it compiles, which of these is the output?

Possible Output

```
File  Edit  Window  Help  Cling
%java StaticTests
static block 4
in main
super static block
super constructor
constructor
```

Possible Output

```
File  Edit  Window  Help  Electricity
%java StaticTests
super static block
static block 3
in main
super constructor
constructor
```

This chapter explored the wonderful, static, world of Java. Your job is to decide whether each of the following statements is true or false.

👍TRUE OR FALSE👎

1. To use the Math class, the first step is to make an instance of it.

2. You can mark a constructor with the **static** keyword.

3. Static methods don't have access to instance variable state of the 'this' object.

4. It is good practice to call a static method using a reference variable.

5. Static variables could be used to count the instances of a class.

6. Constructors are called before static variables are initialized.

7. MAX_SIZE would be a good name for a static final variable.

8. A static initializer block runs before a class's constructor runs.

9. If a class is marked final, all of its methods must be marked final.

10. A final method can only be overridden if its class is extended.

11. There is no wrapper class for boolean primitives.

12. A wrapper is used when you want to treat a primitive like an object.

13. The parseXxx methods always return a String.

14. Formatting classes (which are decoupled from I/O), are in the java.format package.

Lunar Code Magnets

This one might actually be useful! In addition to what you've learned in the last few pages about manipulating dates, you'll need a little more information... First, full moons happen every 29.52 days or so. Second, there was a full moon on Jan. 7th, 2004. Your job is to reconstruct the code snippets to make a working Java program that produces the output listed below (plus more full moon dates). (You might not need all of the magnets, and add all the curly braces you need.) Oh, by the way, your output will be different if you don't live in the mountain time zone.
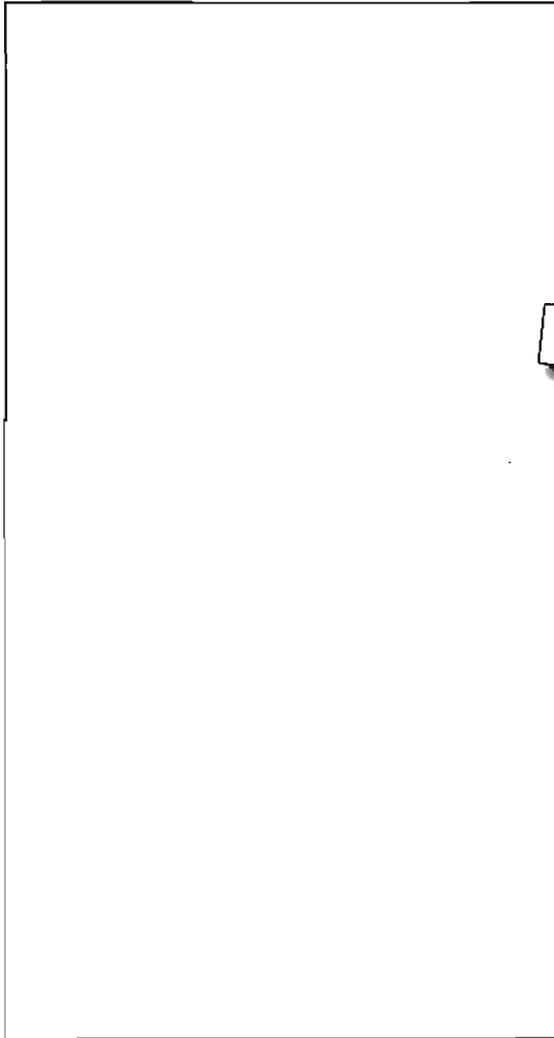
```
long day1 = c.getTimeInMillis();

                          c.set(2004,1,7,15,40);

import static java.lang.System.out;

                static int DAY_IM = 60 * 60 * 24;

("full moon on %tc", c));

                                    (c.format

Calendar c = new Calendar();

                                 class FullMoons {

public static void main(String [] args) {

  day1 += (DAY_IM * 29.52);

                    for (int x = 0; x < 60; x++) {

        static int DAY_IM = 1000 * 60 * 60 * 24;

  println

                          import java.io.*;

("full moon on %t", c));

                    import java.util.*;

static import java.lang.System.out;

        c.set(2004,0,7,15,40);        out.println

c.setTimeInMillis(day1);              (String.format

          Calendar c = Calendar.getInstance();
```

```
File Edit Window Help Howl
% java FullMoons
full moon on Fri Feb 06 04:09:35 MST 2004
full moon on Sat Mar 06 16:38:23 MST 2004
full moon on Mon Apr 05 06:07:11 MDT 2004
```

Exercise Solutions

BE the compiler

```
StaticSuper( ) {
  System.out.println(
    "super constructor");
}
```

StaticSuper is a constructor, and must have ( ) in its signature. Notice that as the output below demonstrates, the static blocks for both classes run before either of the constructors run.

Possible Output

```
File Edit Window Help Cling
%java StaticTests
super static block
static block 3
in main
super constructor
constructor
```

True or False

1. To use the Math class, the first step is to make an instance of it. **False**

2. You can mark a constructor with the keyword 'static'. **False**

3. Static methods don't have access to an object's instance variables. **True**

4. It is good practice to call a static method using a reference variable. **False**

5. Static variables could be used to count the instances of a class. **True**

6. Constructors are called before static variables are initialized. **False**

7. MAX_SIZE would be a good name for a static final variable. **True**

8. A static initializer block runs before a class's constructor runs. **True**

9. If a class is marked final, all of its methods must be marked final. **False**

10. A final method can only be overridden if its class is extended. **False**

11. There is no wrapper class for boolean primitives. **False**

12. A wrapper is used when you want to treat a primitive like an object. **True**

13. The parseXxx methods always return a String. **False**

14. Formatting classes (which are decoupled from I/O), are in the java.format package. **False**

Exercise Solutions

```java
import java.util.*;

import static java.lang.System.out;

class FullMoons {

    static int DAY_IM = 1000 * 60 * 60 * 24;

    public static void main(String [] args) {

        Calendar c = Calendar.getInstance();

        c.set(2004,0,7,15,40);

        long day1 = c.getTimeInMillis();

        for (int x = 0; x < 60; x++) {

            day1 += (DAY_IM * 29.52)

            c.setTimeInMillis(day1);

            out.println(String.format("full moon on %tc", c));

        }

    }

}
```

Notes on the Lunar Code Magnet:

You might discover that a few of the dates produced by this program are off by a day. This astronomical stuff is a little tricky, and if we made it perfect, it would be too complex to make an exercise here.

Hint: one problem you might try to solve is based on differences in time zones. Can you spot the issue?

```
File Edit Window Help Howl
% java FullMoons
full moon on Fri Feb 06 04:09:35 MST 2004
full moon on Sat Mar 06 16:38:23 MST 2004
full moon on Mon Apr 05 06:07:11 MDT 2004
```