1.2 DATA ABSTRACTION

A DATA TYPE is a set of values and a set of operations on those values. So far, we have discussed in detail Java's *primitive* data types: for example, the *values* of the primitive data type `int` are integers between -2^{31} and $2^{31} - 1$; the *operations* of `int` include `+`, `*`, `-`, `/`, `%`, `<`, and `>`. In principle, we could write all of our programs using only the built-in primitive types, but it is much more convenient to write programs at a higher level of abstraction. In this section, we focus on the process of defining and using data types, which is known as *data abstraction* (and supplements the *function abstraction* style that is the basis of SECTION 1.1).

Programming in Java is largely based on building data types known as *reference types* with the familiar Java `class`. This style of programming is known as *object-oriented programming*, as it revolves around the concept of an *object*, an entity that holds a data type value. With Java's primitive types we are largely confined to programs that operate on numbers, but with reference types we can write programs that operate on strings, pictures, sounds, any of hundreds of other abstractions that are available in Java's standard libraries or on our booksite. Even more significant than libraries of predefined data types is that the range of data types available in Java programming is open-ended, because *you can define your own data types* to implement any abstraction whatsoever.

An *abstract data type* (ADT) is a data type whose representation is hidden from the client. Implementing an ADT as a Java class is not very different from implementing a function library as a set of static methods. The primary difference is that we associate *data* with the function implementations and we hide the representation of the data from the client. When *using* an ADT, we focus on the *operations* specified in the API and pay no attention to the data representation; when *implementing* an ADT, we focus on the *data*, then implement operations on that data.

Abstract data types are important because they support encapsulation in program design. In this book, we use them as a means to

- Precisely specify problems in the form of APIs for use by diverse clients
- Describe algorithms and data structures as API implementations

Our primary reason for studying different algorithms for the same task is that performance characteristics differ. Abstract data types are an appropriate framework for the study of algorithms because they allow us to put knowledge of algorithm performance to immediate use: we can substitute one algorithm for another to improve performance for all clients without changing any client code.

Using abstract data types *You do not need to know how a data type is implemented in order to be able to use it*, so we begin by describing how to write programs that use a simple data type named `Counter` whose values are a name and a nonnegative integer and whose operations are *create and initialize to zero*, *increment by one*, and *examine the current value.* This abstraction is useful in many contexts. For example, it would be reasonable to use such a data type in electronic voting software, to ensure that the only thing that a voter can do is increment a chosen candidate's tally by one. Or, we might use a `Counter` to keep track of fundamental operations when analyzing the performance of algorithms. To use a `Counter`, you need to learn our mechanism for specifying the operations defined in the data type and the Java language mechanisms for creating and manipulating data-type values. Such mechanisms are critically important in modern programming, and we use them throughout this book, so this first example is worthy of careful attention.

API for an abstract data type. To specify the behavior of an abstract data type, we use an *application programming interface* (API), which is a list of *constructors* and *instance methods* (operations), with an informal description of the effect of each, as in this API for `Counter`:

`public class Counter`		
	`Counter(String id)`	*create a counter named* `id`
`void`	`increment()`	*increment the counter by one*
`int`	`tally()`	*number of increments since creation*
`String`	`toString()`	*string representation*

An API for a counter

Even though the basis of a data-type definition is a set of values, the role of the values is not visible from the API, only the operations on those values. Accordingly, an ADT definition has many similarities with a library of static methods (see page 24):

- Both are implemented as a Java `class`.
- Instance methods may take zero or more arguments of a specified type, separated by commas and enclosed in parentheses.
- They may provide a return value of a specified type or no return value (signified by `void`).

And there are three significant differences:

- Some entries in the API have the same name as the class and lack a return type. Such entries are known as *constructors* and play a special role. In this case, `Counter` has a constructor that takes a `String` argument.

- Instance methods lack the `static` modifier. They are *not* static methods—their purpose is to operate on data type values.
- Some instance methods are present so as to adhere to Java conventions—we refer to such methods as *inherited methods* and shade them gray in the API.

As with APIs for libraries of static methods, an API for an abstract data type is a contract with all clients and, therefore, the starting point both for developing any client code and for developing any data-type implementation. In this case, the API tells us that to use `Counter`, we have available the `Counter()` constructor, the `increment()` and `tally()` instance methods, and the inherited `toString()` method.

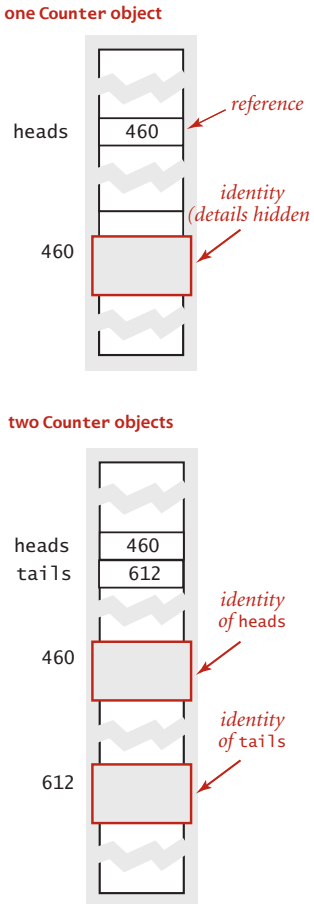
Inherited methods. Various Java conventions enable a data type to take advantage of built-in language mechanisms by including specific methods in the API. For example, all Java data types *inherit* a `toString()` method that returns a `String` representation of the data-type values. Java calls this method when any data-type value is to be concatenated with a `String` value with the + operator. The default implementation is not particularly useful (it gives a string representation of the memory address of the data-type value), so we often provide an implementation that overrides the default, and include `toString()` in the API whenever we do so. Other examples of such methods include `equals()`, `compareTo()`, and `hashCode()` (see page 101).

Client code. As with modular programming based on static methods, the API allows us to write client code without knowing details of the implementation (and to write implementation code without knowing details of any particular client). The mechanisms introduced on page 28 for organizing programs as independent modules are useful for all Java classes, and thus are effective for modular programming with ADTs as well as for libraries of static methods. Accordingly, we can use an ADT in any program provided that the source code is in a `.java` file in the same directory, or in the standard Java library, or accessible through an `import` statement, or through one of the classpath mechanisms described on the booksite. All of the benefits of modular programming follow. By encapsulating all the code that implements a data type within a single Java class, we enable the development of client code at a higher level of abstraction. To develop client code, you need to be able to *declare variables*, *create objects* to hold data-type values, and *provide access* to the values for instance methods to operate on them. These processes are different from the corresponding processes for primitive types, though you will notice many similarities.

Objects. Naturally, you can declare that a variable `heads` is to be associated with data of type `Counter` with the code

```
Counter heads;
```

but how can you assign values or specify operations? The answer to this question involves a fundamental concept in data abstraction: an *object* is an entity that can take on a data-type value. Objects are characterized by three essential properties: *state, identity,* and *behavior*. The *state* of an object is a value from its data type. The *identity* of an object distinguishes one object from another. It is useful to think of an object's identity as the place where its value is stored in memory. The *behavior* of an object is the effect of data-type operations. The implementation has the sole responsibility for maintaining an object's identity, so that client code can use a data type without regard to the representation of its state by conforming to an API that describes an object's behavior. An object's state might be used to provide information to a client or cause a side effect or be changed by one of its data type's operations, but the details of the representation of the data-type value are not relevant to client code. A *reference* is a mechanism for accessing an object. Java nomenclature makes clear the distinction from primitive types (where variables are associated with values) by using the term *reference types* for nonprimitive types. The details of implementing references vary in Java implementations, but it is useful to think of a reference as a memory address, as shown at right (for brevity, we use three-digit memory addresses in the diagram).



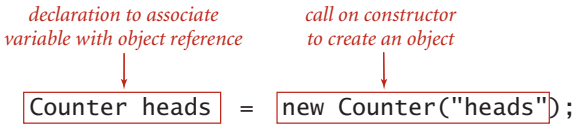
Object representation

Creating objects. Each data-type value is stored in an object. To create (or *instantiate*) an individual object, we invoke a constructor by using the keyword `new`, followed by the class name, followed by `()` (or a list of argument values enclosed in parentheses, if the constructor takes arguments). A constructor has no return type because it always returns a reference to an object of its data type. Each time that a client uses `new()`, the system

- Allocates memory space for the object
- Invokes the constructor to initialize its value
- Returns a reference to the object

In client code we typically create objects in an initializing declaration that associates a variable with the object, as we often do with variables of primitive types. Unlike primitive types, variables are associated with references to objects, not the data-type values

themselves. We can create any number of objects from the same class—each object has its own identity and may or may not store the same value as another object of the same type. For example, the code

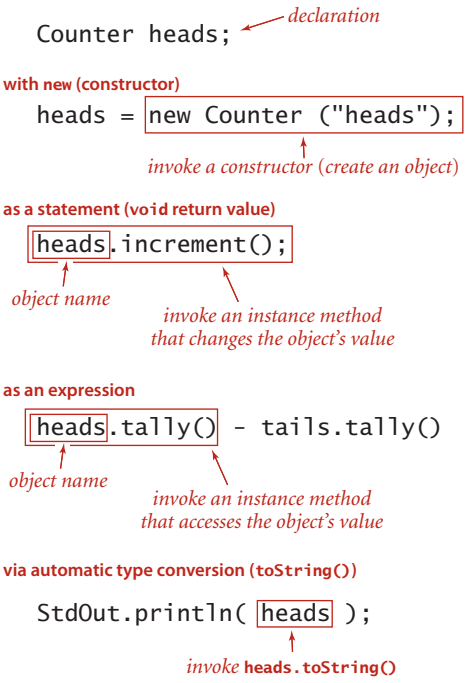


Creating an object

```
Counter heads = new Counter("heads");
Counter tails = new Counter("tails");
```

creates two different `Counter` objects. In an abstract data type, details of the representation of the value are hidden from client code. You might assume that the value associated with each `Counter` object is a `String` name and an `int` tally, but *you cannot write code that depends on any specific representation* (or even know whether that assumption is true—perhaps the tally is a `long` value).

Invoking instance methods. The purpose of an instance method is to operate on data-type values, so the Java language includes a special mechanism to invoke instance methods that emphasizes a connection to an object. Specifically, we invoke an instance method by writing a variable name that refers to an object, followed by a period, followed by an instance method name, followed by 0 or more arguments, enclosed in parentheses and separated by commas. An instance method might *change* the data-type value or just *examine* the data-type value. Instance methods have all of the properties of static methods that we considered on page 24—arguments are passed by value, method names can be overloaded, they may have a return value, and they may cause side effects—but they have an additional property that characterizes them: *each invocation is associated with an object*. For example, the code



Invoking instance methods

```
heads.increment();
```

invokes the instance method `increment()` to operate on the `Counter` object `heads` (in this case the operation involves incrementing the tally), and the code

```
heads.tally() - tails.tally();
```

invokes the instance method `tally()` twice, first to operate on the `Counter` object `heads` and then to operate on the `Counter` object `tails` (in this case the

operation involves returning the tally as an `int` value). As these examples illustrate, you can use calls on instance methods in client code in the same way as you use calls on static methods—as statements (`void` methods) or values in expressions (methods that return a value). The primary purpose of static methods is to implement functions; the primary purpose of non-static (instance) methods is to implement data-type operations. Either type of method may appear in client code, but you can easily distinguish between them, because a static method call starts with a *class* name (uppercase, by convention) and a non-static method call always starts with an *object* name (lowercase, by convention). These differences are summarized in the table at right.

	instance method	static method
sample call	`head.increment()`	`Math.sqrt(2.0)`
invoked with	object name	class name
parameters	reference to object and argument(s)	argument(s)
primary purpose	examine or change object value	compute return value

Instance methods versus static methods

Using objects. Declarations give us variable names for objects that we can use in code not just to create objects and invoke instance methods, but also in the same way as we use variable names for integers, floating-point numbers, and other primitive types. To develop client code for a given data type, w:

- Declare variables of the type, for use in referring to objects
- Use the keyword `new` to invoke a constructor that creates objects of the type
- Use the object name to invoke instance methods, either as statements or within expressions

For example, the class `Flips` shown at the top of the next page is a `Counter` client that takes a command-line argument `T` and simulates `T` coin flips (it is also a `StdRandom` client). Beyond these direct uses, we can use variables associated with objects in the same way as we use variables associated with primitive-type values:

- In assignment statements
- To pass or return objects from methods
- To create and use arrays of object.

Understanding the behavior of each of these types of uses requires thinking in terms of *references*, not values, as you will see when we consider them, in turn.

Assignment statements. An assignment statement with a reference type creates a copy of the reference. The assignment statement does not create a new object, just another reference to an existing object. This situation is known as *aliasing*: both variables refer to the same object. The effect of aliasing is a bit unexpected, because it is different for variables holding values of a primitive type. Be sure that you understand the difference.

```
public class Flips
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      Counter heads = new Counter("heads");
      Counter tails = new Counter("tails");
      for (int t = 0; t < T; t++)
         if (StdRandom.bernoulli(0.5))
              heads.increment();
         else tails.increment();
      StdOut.println(heads);
      StdOut.println(tails);
      int d = heads.tally() - tails.tally();
      StdOut.println("delta: " + Math.abs(d));
   }
}
```

```
% java Flips 10
5 heads
5 tails
delta: 0

% java Flips 10
8 heads
2 tails
delta: 6

% java Flips 1000000
499710 heads
500290 tails
delta: 580
```

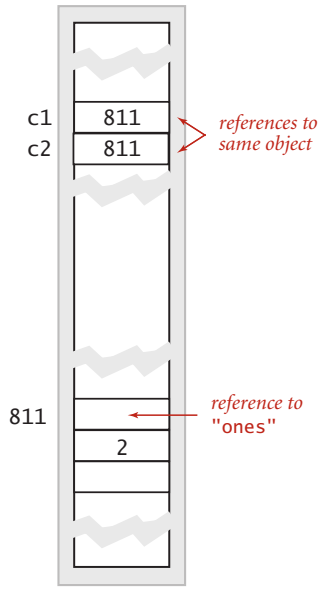
Counter client that simulates T coin flips

If x and y are variables of a primitive type, then the assignment `x = y` copies the value of y to x. For reference types, the *reference* is copied (not the value). Aliasing is a common source of bugs in Java programs, as illustrated by the following example:

```
Counter c1 = new Counter("ones");
c1.increment();
Counter c2 = c1;
c2.increment();
StdOut.println(c1);
```

With a typical `toString()` implementation this code would print the string `"2 ones"` which may or may not be what was intended and is counterintuitive at first. Such bugs are common in programs written by people without much experience in using objects (that may be you, so pay attention here!). Changing the state of an object impacts all code involving aliased variables referencing that object. We are used to thinking of two different variables of primitive types as being independent, but that intuition does not carry over to variables of reference types.

```
Counter c1;
c1 = new Counter("ones");
c1.increment();
Counter c2 = c1;
c2.increment();
```



Aliasing

Objects as arguments. You can pass objects as *arguments* to methods. This ability typically simplifies client code. For example, when we use a `Counter` as an argument, we are essentially passing both a name and a tally, but need only specify one variable. When we call a method with arguments, the effect in Java is as if each argument value were to appear on the right-hand side of an assignment statement with the corresponding argument name on the left. That is, Java passes a *copy* of the argument value from the calling program to the method. This arrangement is known as *pass by value* (see page 24). One important consequence is that the method cannot change the value of a caller's variable. For primitive types, this policy is what we expect (the two variables are independent), but each time that we use a reference type as a method argument we create an alias, so we must be cautious. In other words, the convention is to pass the *reference* by value (make a copy of it) but to pass the *object* by reference. For example, if we pass a reference to an object of type `Counter`, the method cannot change the original reference (make it point to a different `Counter`), but it *can* change the value of the object, for example by using the reference to call `increment()`.

Objects as return values. Naturally, you can also use an object as a *return value* from a method. The method might return an object passed to it as an argument, as in the example below, or it might create an object and return a reference to it. This capability is important because Java methods allow only one return value—using objects enables us to write code that, in effect, returns multiple values.

```
% java FlipsMax 1000000
500281 tails wins
```

```
public class FlipsMax
{
   public static Counter max(Counter x, Counter y)
   {
      if (x.tally() > y.tally()) return x;
      else                       return y;
   }

   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      Counter heads = new Counter("heads");
      Counter tails = new Counter("tails");
      for (int t = 0; t < T; t++)
         if (StdRandom.bernoulli(0.5))
              heads.increment();
         else tails.increment();

      if (heads.tally() == tails.tally())
           StdOut.println("Tie");
      else StdOut.println(max(heads, tails) + " wins");
   }
}
```

Example of a static method with object arguments and return values

Arrays are objects. In Java, every value of any nonprimitive type is an object. In particular, arrays are objects. As with strings, there is special language support for certain operations on arrays: declarations, initialization, and indexing. As with any other object, when we pass an array to a method or use an array variable on the right hand side of an assignment statement, we are making a copy of the array reference, not a copy of the array. This convention is appropriate for the typical case where we expect the method to be able to modify the array, by rearranging its entries, as, for example, in `java.util.Arrays.sort()` or the `shuffle()` method that we considered on page 32.

Arrays of objects. Array entries can be of any type, as we have already seen: `args[]` in our `main()` implementations is an array of `String` objects. When we create an array of objects, we do so in two steps:

- Create the array, using the bracket syntax for array constructors.
- Create each object in the array, using a standard constructor for each.

For example, the code below simulates rolling a die, using an array of `Counter` objects to keep track of the number of occurrences of each possible value. An array of objects in Java is an array of references to objects, not the objects themselves. If the objects are large, then we may gain efficiency by not having to move them around, just their references. If they are small, we may lose efficiency by having to follow a reference each time we need to get to some information.

```
public class Rolls
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      int SIDES = 6;
      Counter[] rolls = new Counter[SIDES+1];
      for (int i = 1; i <= SIDES; i++)
         rolls[i] = new Counter(i + "'s");

      for (int t = 0; t < T; t++)
      {
         int result = StdRandom.uniform(1, SIDES+1);
         rolls[result].increment();
      }
      for (int i = 1; i <= SIDES; i++)
         StdOut.println(rolls[i]);
   }
}
```

```
% java Rolls 1000000
167308 1's
166540 2's
166087 3's
167051 4's
166422 5's
166592 6's
```

Counter client that simulates T rolls of a die

WITH THIS FOCUS ON OBJECTS, writing code that embraces data abstraction (defining and using data types, with data-type values held in objects) is widely referred to as *object-oriented programming*. The basic concepts that we have just covered are the starting point for object-oriented programming, so it is worthwhile to briefly summarize them. A *data type* is a set of values and a set of operations defined on those values. We implement data types in independent Java `class` modules and write client programs that use them. An *object* is an entity that can take on a data-type value or an *instance* of a data type. Objects are characterized by three essential properties: *state, identity,* and *behavior*. A data-type implementation supports clients of the data type as follows:

- Client code can *create objects* (establish identity) by using the `new` construct to invoke a constructor that creates an object, initializes its instance variables, and returns a reference to that object.
- Client code can *manipulate data-type values* (control an object's behavior, possibly changing its state) by using a variable associated with an object to invoke an instance method that operates on that object's instance variables.
- Client code can *manipulate objects* by creating arrays of objects and passing them and returning them to methods, in the same way as for primitive-type values, except that variables refer to references to values, not the values themselves.

These capabilities are the foundation of a flexible, modern, and widely useful programming style that we will use as the basis for studying algorithms in this book.

Examples of abstract data types The Java language has thousands of built-in ADTs, and we have defined many other ADTs to facilitate the study of algorithms. Indeed, every Java program that we write is a data-type implementation (or a library of static methods). To control complexity, we will specifically cite APIs for any ADT that we use in this book (not many, actually).

In this section, we introduce as examples several data types, with some examples of client code. In some cases, we present excerpts of APIs that may contain dozens of instance methods or more. We articulate these APIs to present real-world examples, to specify the instance methods that we will use in the book, and to emphasize that you do not need to know the details of an ADT implementation in order to be able to use it.

For reference, the data types that we use and develop in this book are shown on the facing page. These fall into several different categories:

- Standard system ADTs in `java.lang.*`, which can be used in any Java program.
- Java ADTs in libraries such as `java.awt`, `java.net`, and `java.io`, which can also be used in any Java program, but need an `import` statement.
- Our I/O ADTs that allow us to work with multiple input/output streams similar to `StdIn` and `StdOut`.
- Data-oriented ADTs whose primary purpose is to facilitate organizing and processing data by encapsulating the representation. We describe several examples for applications in computational geometry and information processing later in this section and use them as examples in client code later on.
- Collection ADTs whose primary purpose is to facilitate manipulation collections of data of the same. We describe the basic `Bag`, `Stack`, and `Queue` types in SECTION 1.3, `PQ` types in CHAPTER 2, and the `ST` and `SET` types in CHAPTERS 3 and 5.
- Opertions-oriented ADTs that we use to analyze algorithms, as described in SECTION 1.4 and SECTION 1.5.
- ADTs for graph algorithms, including both data-oriented ADTs that focus on encapsulating representations of various kinds of graphs and operations-oriented ADTs that focus on providing specifications for graph-processing algorithms.

This list does not include the dozens of types that we consider in exercises, which may be found in the index. Also, as described on page 90, we often distinguish multiple implementations of various ADTs with a descriptive prefix. As a group, the ADTs that we use demonstrate that organizing and understanding the data types that you use is an important factor in modern programming.

A typical application might use only five to ten of these ADTs. A prime goal in the development and organization of the ADTs in this book is to enable programmers to easily take advantage of a relatively small set of them in developing client code.

standard Java system types in `java.lang`

`Integer`	`int` *wrapper*
`Double`	`double` *wrapper*
`String`	*indexed* `char`*s*
`StringBuilder`	*builder for strings*

other Java types

`java.awt.Color`	*colors*
`java.awt.Font`	*fonts*
`java.net.URL`	*URLs*
`java.io.File`	*files*

our standard I/O types

`In`	*input stream*
`Out`	*output stream*
`Draw`	*drawing*

data-oriented types for client examples

`Point2D`	*point in the plane*
`Interval1D`	*1D interval*
`Interval2D`	*2D interval*
`Date`	*date*
`Transaction`	*transaction*

types for the analysis of algorithms

`Counter`	*counter*
`Accumulator`	*accumulator*
`VisualAccumulator`	*visual version*
`Stopwatch`	*stopwatch*

collection types

`Stack`	*pushdown stack*
`Queue`	*FIFO queue*
`Bag`	*bag*
`MinPQ MaxPQ`	*priority queue*
`IndexMinPQ IndexMinPQ`	*priority queue (indexed)*
`ST`	*symbol table*
`SET`	*set*
`StringST`	*symbol table (string keys)*

data-oriented graph types

`Graph`	*graph*
`Digraph`	*directed graph*
`Edge`	*edge (weighted)*
`EdgeWeightedGraph`	*graph (weighted)*
`DirectedEdge`	*edge (directed, weighted)*
`EdgeWeightedDigraph`	*graph (directed, weighted)*

operations-oriented graph types

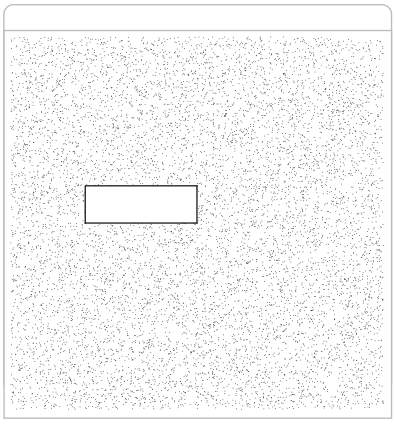
`UF`	*dynamic connectivity*
`DepthFirstPaths`	*DFS path searcher*
`CC`	*connected components*
`BreadthFirstPaths`	*BFS path search*
`DirectedDFS`	*DFS digraph path search*
`DirectedBFS`	*BFS digraph path search*
`TransitiveClosure`	*all paths*
`Topological`	*topological order*
`DepthFirstOrder`	*DFS order*
`DirectedCycle`	*cycle search*
`SCC`	*strong components*
`MST`	*minimum spanning tree*
`SP`	*shortest paths*

Selected ADTs used in this book

Geometric objects. A natural example of object-oriented programming is designing data types for geometric objects. For example, the APIs on the facing page define abstract data types for three familiar geometric objects: `Point2D` (points in the plane), `Interval1D` (intervals on the line), and `Interval2D` (two-dimensional intervals in the plane, or axis-aligned rectangles). As usual, the APIs are essentially self-documenting and lead immediately to easily understood client code such as the example at left, which reads the boundaries of an `Interval2D` and an integer T from the command line, generates T random points in the unit square, and counts the number of points that fall in the interval (an estimate of the area of the rectangle). For dramatic effect, the client also draws the interval and the points that fall outside the interval. This computation is a model for a method that reduces the problem of computing the area and volume of geometric shapes to the problem of determining whether a point falls within the shape or not (a less difficult but not trivial problem). Of course, we can define APIs for other geometric objects such as line segments, triangles, polygons, circles, and so forth, though implementing operations on them can be challenging. Several examples are addressed in the exercises at the end of this section.

```
public static void main(String[] args)
{
   double xlo = Double.parseDouble(args[0]);
   double xhi = Double.parseDouble(args[1]);
   double ylo = Double.parseDouble(args[2]);
   double yhi = Double.parseDouble(args[3]);
   int T = Integer.parseInt(args[4]);

   Interval1D x = new Interval1D(xlo, xhi);
   Interval1D y = new Interval1D(ylo, yhi);
   Interval2D box = new Interval2D(x, y);
   box.draw();

   Counter c = new Counter("hits");
   for (int t = 0; t < T; t++)
   {
      double x = Math.random();
      double y = Math.random();
      Point p = new Point(x, y);
      if (box.contains(p)) c.increment();
      else                 p.draw();
   }

   StdOut.println(c);
   StdOut.println(box.area());
}
```

Interval2D test client



```
% java Interval2D .2 .5 .5 .6 10000
297 hits
.03
```

PROGRAMS THAT PROCESS GEOMETRIC OBJECTS have wide application in computing with models of the natural world, in scientific computing, video games, movies, and many other applications. The development and study of such programs and applications has blossomed into a far-reaching field of study known as *computational geometry*, which is a

public class Point2D	
Point2D(double x, double y)	*create a point*
double x()	*x coordinate*
double y()	*y coordinate*
double r()	*radius (polar coordinates)*
double theta()	*angle (polar coordinates)*
double distTo(Point2D that)	*Euclidean distance from this point to* that
void draw()	*draw the point on* StdDraw

An API for points in the plane

public class Interval1D	
Interval1D(double lo, double hi)	*create an interval*
double length()	*length of the interval*
boolean contains(double x)	*does the interval contain* x*?*
boolean intersects(Interval1D that)	*does the interval intersect* that*?*
void draw()	*draw the interval on* StdDraw

An API for intervals on the line

public class Interval2D	
Interval2D(Interval1D x, Interval1D y)	*create a 2D interval*
double area()	*area of the 2D interval*
boolean contains(Point p)	*does the 2D interval contain* p*?*
boolean intersects(Interval2D that)	*does the 2D interval intersect* that*?*
void draw()	*draw the 2D interval on* StdDraw

An API for two dimensional intervals in the plane

fertile area of examples for the application of the algorithms that we address in this book, as you will see in examples throughout the book. In the present context, our interest is to suggest that abstract data types that directly represent geometric abstractions are not difficult to define and can lead to simple and clear client code. This idea is reinforced in several exercises at the end of this section and on the booksite.

Information processing Whether it be a bank processing millions of credit card transactions or a web analytics company processing billions of touchpad taps or a scientific research group processing millions of experimental observations, a great many applications are centered around processing and organizing information. Abstract data types provide a natural mechanism for organizing the information. Without getting into details, the two APIs on the facing page suggest a typical approach for a commercial application. The idea is to define data types that allow us to keep information in objects that correspond to things in the real world. A date is a day, a month, and a year and a transaction is a customer, a date, and an amount. These two are just examples: we might also define data types that can hold detailed information for customers, times, locations, goods and services, or whatever. Each data type consists of constructors that create objects containing the data and methods for use by client code to access it. To simplify client code, we provide two constructors for each type, one that presents the data in its appropriate type and another that parses a string to get the data (see EXERCISE 1.2.19 for details). As usual, there is no reason for client code to know the representation of the data. Most often, the reason to organize the data in this way is to treat the data associated with an object as a single entity: we can maintain arrays of `Transaction` values, use `Date` values as a argument or a return value for a method, and so forth. The focus of such data types is on encapsulating the data, while at the same time enabling the development of client code that does not depend on the representation of the data. We do not dwell on organizing information in this way, except to take note that doing so and including the inherited methods `toString()`, `compareTo()`, `equals()`, and `hashCode()` allows us to take advantage of algorithm implementations that can process *any type of data*. We will discuss inherited methods in more detail on page 100. For example, we have already noted Java's convention that enables clients to print a string representation of every value if we include `toString()` implementation in a data type. We consider conventions corresponding to the other inherited methods in SECTION 1.3, SECTION 2.5, SECTION 3.4, and SECTION 3.5, using `Date` and `Transaction` as examples. SECTION 1.3 gives classic examples of data types and a Java language mechanism known as *parameterized types,* or *generics*, that takes advantage of these conventions, and CHAPTER 2 and CHAPTER 3 are also devoted to taking advantage of generic types and inherited methods to develop implementations of sorting and searching algorithms that are effective for any type of data.

WHENEVER YOU HAVE DATA OF DIFFERENT TYPES that logically belong together, it is worthwhile to contemplate defining an ADT as in these examples. The ability to do so helps to organize the data, can greatly simplify client code in typical applications, and is an important step on the road to data abstraction.

public class Date implements Comparable<Date>	
Date(int month, int day, int year)	*create a date*
Date(String date)	*create a date (parse constructor)*
int month()	*month*
int day()	*day*
int year()	*year*
String toString()	*string representation*
boolean equals(Object that)	*is this the same date as* that*?*
int compareTo(Date that)	*compare this date to* that
int hashCode()	*hash code*

public class Transaction implements Comparable<Transaction>	
Transaction(String who, Date when, double amount)	
Transaction(String transaction)	*create a transaction (parse constructor)*
String who()	*customer name*
Date when()	*date*
double amount()	*amount*
String toString()	*string representation*
boolean equals(Object that)	*is this the same transaction as* that*?*
int compareTo(Transaction that)	*compare this transaction to* that
int hashCode()	*hash code*

Sample APIs for commercial applications (dates and transactions)

Strings. Java's `String` is an important and useful ADT. A `String` is an indexed sequence of `char` values. `String` has dozens of instance methods, including the following:

public class String	
`String()`	*create an empty string*
`int length()`	*length of the string*
`int charAt(int i)`	*`i`th character*
`int indexOf(String p)`	*first occurrence of `p` (`-1` if none)*
`int indexOf(String p, int i)`	*first occurrence of `p` after `i` (`-1` if none)*
`String concat(String t)`	*this string with `t` appended*
`String substring(int i, int j)`	*substring of this string (`i`th to `j-1`st chars)*
`String[] split(String delim)`	*strings between occurrences of `delim`*
`int compareTo(String t)`	*string comparison*
`boolean equals(String t)`	*is this string's value the same as `t`'s ?*
`int hashCode()`	*hash code*

Java `String` API (partial list of methods)

`String` values are similar to arrays of characters, but the two are not the same. Arrays have built-in Java language syntax for accessing a character; `String` has instance methods for indexed access, length, and many other operations. On the other hand, `String` has special language support for initialization and concatenation: instead of creating and initializing a string with a constructor, we can use a string literal; instead of invoking the method `concat()` we can use the + operator. We do not need to consider the details of the implementation, though understanding performance characteristics of some of the methods is important when developing string-processing algorithms, as you will see in CHAPTER 5. Why not just use arrays of characters instead of `String` values? The answer to this question is the same as for any ADT: *to simplify and clarify client code*. With `String`, we can write clear and simple client code that uses numerous convenient instance methods without regard to the way in which strings are represented (see facing page). Even this short list contains powerful operations that require advanced algorithms such

```
String a = "now is ";
String b = "the time ";
String c = "to"
```

call	*value*
`a.length()`	`7`
`a.charAt(4)`	`i`
`a.concat(c)`	`"now is to"`
`a.indexOf("is")`	`4`
`a.substring(2, 5)`	`"w i"`
`a.split(" ")[0]`	`"now"`
`a.split(" ")[1]`	`"is"`
`b.equals(c)`	`false`

Examples of string operations

task	implementation

is the string a palindrome?	`public static boolean isPalindrome(String s)`
	`{`
	`  int N = s.length();`
	`  for (int i = 0; i < N/2; i++)`
	`  if (s.charAt(i) != s.charAt(N-1-i))`
	`  return false;`
	`  return true;`
	`}`

extract file name and extension from a command-line argument	`String s = args[0];`
	`int dot = s.rank(".");`
	`String base  = s.substring(0, dot);`
	`String extension = s.substring(dot + 1, s.length());`

print all lines in standard input that contain a string specified on the command line	`String query = args[0];`
	`while (!StdIn.isEmpty())`
	`{`
	`  String s = StdIn.readLine();`
	`  if (s.contains(query)) StdOut.println(s);`
	`}`

create an array of the strings on `StdIn` *delimited by whitespace*	`String input = StdIn.readAll();` `String[] words = input.split("\\s+");`

check whether an array of strings is in alphabetical order	`public boolean isSorted(String[] a)`
	`{`
	`  for (int i = 1; i < a.length; i++)`
	`  {`
	`  if (a[i-1].compareTo(a[i]) > 0)`
	`  return false;`
	`  }`
	`  return true;`
	`}`

Typical string-processing code

as those considered in CHAPTER 5. For example, the argument of `split()` can be a *regular expression* (see SECTION 5.4)—the `split()` example on page 81 uses the argument "\\s+", which means "one or more tabs, spaces, newlines, or returns."

Input and output revisited. A disadvantage of the `StdIn`, `StdOut`, and `StdDraw` standard libraries of SECTION 1.1 is that they restrict us to working with just one input file, one output file, and one drawing for any given program. With object-oriented programming, we can define similar mechanisms that allow us to work with *multiple* input streams, output streams, and drawings within one program. Specifically, our standard libary includes the data types `In`, `Out`, and `Draw` with the APIs shown on the facing page, When invoked with a constructor having a `String` argument, `In` and `Out` will first try to find a file in the current directory of your computer that has that name. If it cannot do so, it will assume the argument to be a website name and will try to connect to that website (if no such website exists, it will issue a runtime exception). In either case, the specified file or website becomes the source/target of the input/output for the stream object thus created, and the `read*()` and `print*()` methods will refer to that file or website. (If you use the no-argument constructor, then you obtain the standard streams.) This arrangement makes it possible for a single program to process multiple files and drawings. You also can assign such objects to variables, pass them as arguments or return values from methods, create arrays of them, and manipulate them just as you manipulate objects of any type. The program `Cat` shown at left is a sample client of `In` and `Out` that uses multiple input streams to concatenate several input files into a single output file. The `In` and `Out` classes also contain static methods for reading files containing values that are all `int`, `double`, or `String` types into an array (see page 126 and EXERCISE 1.2.15).

```
public class Cat
{
   public static void main(String[] args)
   {  // Copy input files to out (last argument).
      Out out = new Out(args[args.length-1]);
      for (int i = 0; i < args.length - 1; i++)
      {  // Copy input file named on ith arg to out.
         In in = new In(args[i]);
         String s = in.readAll();
         out.println(s);
         in.close();
      }
      out.close();
   }
}
```

A sample `In` and `Out` client

```
% more in1.txt
This is

% more in2.txt
a tiny
test.

% java Cat in1.txt in2.txt out.txt

% more out.txt
This is
a tiny
test.
```

public class In	
In()	*create an input stream from standard input*
In(String name)	*create an input stream from a file or website*
boolean isEmpty()	`true` *if no more input,* `false` *otherwise*
int readInt()	*read a value of type* `int`
double readDouble()	*read a value of type* `double`
...	
void close()	*close the input stream*

Note: all operations supported by `StdIn` *are also supported for* `In` *objects.*

API for our data type for input streams

public class Out	
Out()	*create an output stream to standard output*
Out(String name)	*create an output stream to a file*
void print(String s)	*append* `s` *to the output stream*
void println(String s)	*append* `s` *and a newline to the output stream*
void println()	*append a newline to the output stream*
void printf(String f, ...)	*formatted print to the output steam*
void close()	*close the output stream*

Note: all operations supported by `StdOut` *are also supported for* `Out` *objects.*

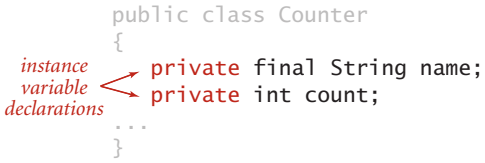
API for our data type for output streams

public class Draw	
Draw()	
void line(double x0, double y0, double x1, double y1)	
void point(double x, double y)	
...	

Note: all operations supported by `StdDraw` *are also supported for* `Draw` *objects.*

API for our data type for drawings

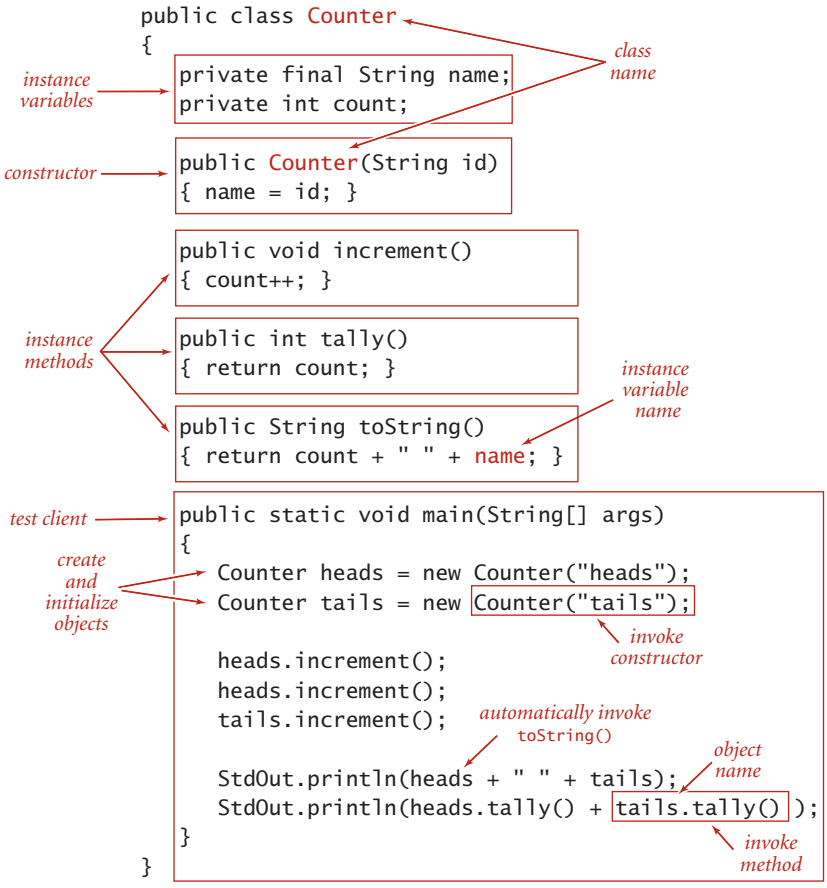
Implementing an abstract data type. As with libraries of static methods, we implement ADTs with a Java `class`, putting the code in a file with the same name as the class, followed by the `.java` extension. The first statements in the file declare *instance variables* that define the data-type values. Following the instance variables are the *constructor* and the *instance methods* that implement operations on data-type values. Instance methods may be *public* (specified in the API) or *private* (used to organize the computation and not available to clients). A data-type definition may have multiple constructors and may also include definitions of static methods. In particular, a unit-test client `main()` is normally useful for testing and debugging. As a first example, we consider an implementation of the `Counter` ADT that we defined on page 65. A full annotated implementation is shown on the facing page, for reference as we discuss its constituent parts. Every ADT implementation that you will develop has the same basic ingredients as this simple example.



Instance variables in ADTs are `private`

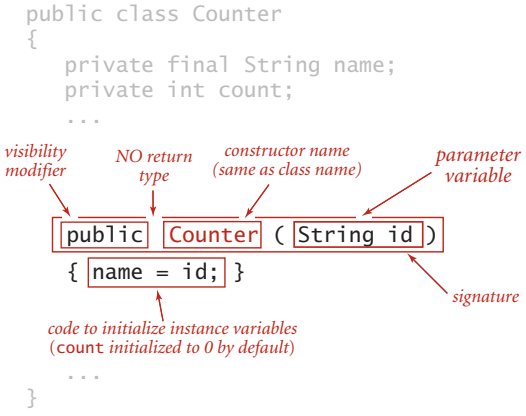
Instance variables. To define data-type values (the *state* of each object), we declare *instance variables* in much the same way as we declare local variables. There is a critical distinction between instance variables and the local variables within a static method or a block that you are accustomed to: there is just *one* value corresponding to each local variable at a given time, but there are *numerous* values corresponding to each instance variable (one for each object that is an instance of the data type). There is no ambiguity with this arrangement, because each time that we access an instance variable, we do so with an object name—that object is the one whose value we are accessing. Also, each declaration is qualified by a *visibility modifier*. In ADT implementations, we use `private`, using a Java language mechansim to enforce the idea that the representation of an ADT is to be hidden from the client, and also `final`, if the value is not to be changed once it is initialized. `Counter` has two instance variables: a `String` value `name` and an `int` value `count`. If we were to use `public` instance variables (allowed in Java) the data type would, by definition, not be abstract, so we do not do so.

Constructors. Every Java class has at least one *constructor* that establishes an object's *identity*. A constructor is like a static method, but it can refer directly to instance variables and has no return value. Generally, the purpose of a constructor is to initialize the instance variables. Every constructor creates an object and provides to the client a reference to that object. Constructors always share the same name as the class. We can overload the name and have multiple constructors with different signatures, just as with methods. If no other constructor is defined, a default no-argument constructor is



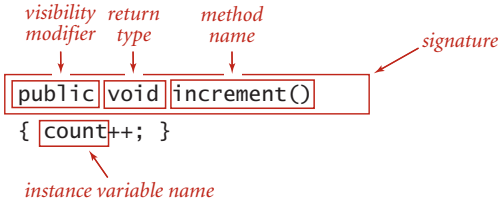
Anatomy of a class that defines a data type

implicit, has no arguments, and initializes instance values to default values. The default values of instance variables are `0` for primitive numeric types, `false` for `boolean`, and `null` for reference types. These defaults may be changed by using initializing declarations for instance variables. Java automatically invokes a constructor when a client program uses the keyword `new`. Overloaded constructors are typically used to initialize instance variables to client-supplied values other than the defaults. For example, `Counter` has a one-argument constructor that initializes the `name` instance variable to the value given as argument (leaving the `count` instance variable to be initialized to the default value `0`).



Anatomy of a constructor

Instance methods. To implement data-type instance methods (the *behavior* of each object), we implement *instance methods* with code that is precisely like the code that you learned in SECTION 1.1 to implement static methods (functions). Each instance method has a return type, a *signature* (which specifies its name and the types and names of its parameter variables), and a *body* (which consists of a sequence of statements, including a *return* statement that provides a value of the return type back to the client). When a client invokes a method, the parameter values (if any) are initialized with client values, the statements are executed until a return value is computed, and the value is returned to the client, with the same effect as if the method invocation in the client were replaced with that value. All of this action is the same as for static methods, but there is one critical distinction for instance methods: *they can access and perform operations on instance variables*. How do we specify which object's instance variables we want to use? If you think about this question for a moment, you will see the logical answer: a reference to a variable in an instance method refers to the value *for the object that was used to invoke the method*. When we say `heads.increment()` the code in `increment()` is referring to the instance variables for `heads`. In other words,



Anatomy of an instance method

object-oriented programming adds one critically important additional way to use variables in a Java program:

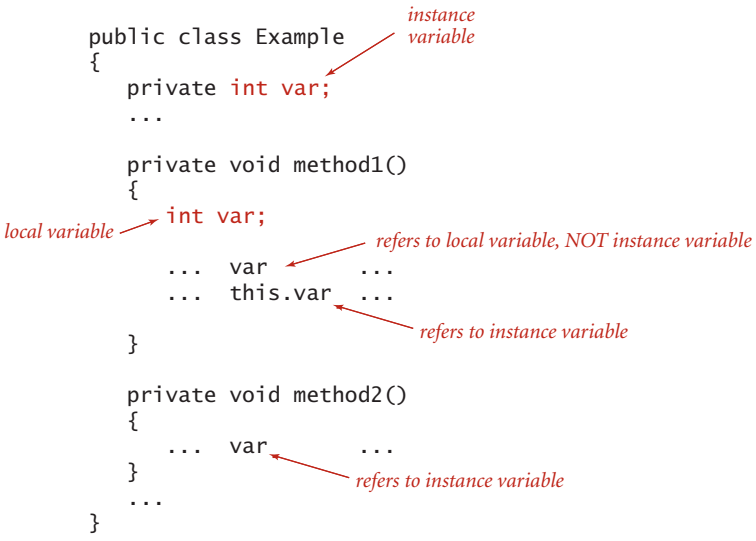
- to invoke an instance method that operates on the object's values.

The difference from working solely with static methods is semantic (see the Q&A), but has reoriented the way that modern programmers think about developing code in many situations. As you will see, it also dovetails well with the study of algorithms and data structures.

Scope. In summary, the Java code that we write to implement instance methods uses *three* kinds of variables:

- Parameter variables
- Local variables
- *Instance variables*

The first two of these are the same as for static methods: parameter variables are specified in the method signature and initialized with client values when the method is called, and local variables are declared and initialized within the method body. The scope of parameter variables is the entire method; the scope of local variables is the following statements in the block where they are defined. Instance variables are completely different: they hold data-type values for objects in a class, and their scope is the entire class (whenever there is an ambiguity, you can use the `this` prefix to identify instance variables). Understanding the distinctions among these three kinds of variables in instance methods is a key to success in object-oriented programming.



Scope of instance and local variables in an instance method

API, clients, and implementations. These are the basic components that you need to understand to be able to build and use abstract data types in Java. Every ADT implementation that we will consider will be a Java class with private instance variables, constructors, instance methods, and a client. To fully understand a data type, we need the API, typical client code, and an implementation, summarized for `Counter` on the facing page. To emphasize the separation of client and implementation, we normally present each client as a separate class containing a static method `main()` and reserve test client's `main()` in the data-type definition for minimal unit testing and development (calling each instance method at least once). In each data type that we develop, we go through the same steps. Rather than thinking about what action we need to take next to accomplish a computational goal (as we did when first learning to program), we think about the needs of a client, then accommodate them in an ADT, following these three steps:

- Specify an API. The purpose of the API is to *separate clients from implementations*, to enable modular programming. We have two goals when specifying an API. First, we want to enable clear and correct client code. Indeed, it is a good idea to write some client code before finalizing the API to gain confidence that the specified data-type operations are the ones that clients need. Second, we want to be able to implement the operations. There is no point specifying operations that we have no idea how to implement.
- Implement a Java class that meets the API specifications. First we choose the instance variables, then we write constructors and the instance methods.
- Develop multiple test clients, to validate the design decisions made in the first two steps.

What operations do clients need to perform, and what data-type values can best support those operations? These basic decisions are at the heart of every implementation that we develop.

API

public class Counter	
Counter(String id)	*create a counter named* id
void increment()	*increment the counter*
int tally()	*number of increments since creation*
String toString()	*string representation*

typical client

```
public class Flips
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);

      Counter heads = new Counter("heads");
      Counter tails = new Counter("tails");

      for (int t = 0; t < T; t++)
         if (StdRandom.bernoulli(0.5))
              heads.increment();
         else tails.increment();

      StdOut.println(heads);
      StdOut.println(tails);
      int d = heads.tally() - tails.tally();
      StdOut.println("delta: " + Math.abs(d));
   }
}
```

implementation

```
public class Counter
{
   private final String name;
   private int count;

   public Counter(String id)
   { name = id; }

   public void increment()
   { count++; }

   public int tally()
   { return count; }

   public String toString()
   { return count + " " + name; }

}
```

application

```
% java Flips 1000000
500172 heads
499828 tails
delta: 344
```

An abstract data type for a simple counter

More ADT implementations As with any programming concept, the best way to understand the power and utility of ADTs is to consider carefully more examples and more implementations. There will be ample opportunity for you to do so, as much of this book is devoted to ADT implementations, but a few more simple examples will help us lay the groundwork for addressing them.

Date. Shown on the facing page are two implementations of the `Date` ADT that we considered on page 79. To reduce clutter, we omit the parsing constructor (which is described in EXERCISE 1.2.19) and the inherited methods `equals()` (see page 103), `compareTo()` (see page 247), and `hashCode()` (see EXERCISE 3.4.22). The straightforward implementation on the left maintains the day, month, and year as instance variables, so that the instance methods can just return the appropriate value; the more space-efficient implementation on the right uses only a single `int` value to represent a date, using a mixed-radix number that represents the date with day d, month m, and year y as $512y + 32m + d$. One way that a client might notice the difference between these implementations is by violating implicit assumptions: the second implementation depends for its correctness on the day being between 0 and 31, the month being between 0 and 15, and the year being positive (in practice, both implementations should check that months are between 1 and 12, days are between 1 and 31, and that dates such as June 31 and February 29, 2009, are illegal, though that requires a bit more work). This example highlights the idea that we rarely *fully* specify implementation requirements in an API (we normally do the best we can, and could do better here). Another way that a client might notice the difference between the two implementations is *performance:* the implementation on the right uses less space to hold data-type values at the cost of more time to provide them to the client in the agreed form (one or two arithmetic operations are needed). Such tradeoffs are common: one client may prefer one of the implementations and another client might prefer the other, so we need to accommodate both. Indeed, one of the recurring themes of this book is that we need to understand the space and time requirements of various implementations and their suitability for use by various clients. One of the key advantages of using data abstraction in our implementations is that we can normally change from one implementation to another *without changing any client code.*

Maintaining multiple implementations. Multiple implementations of the same API can present maintainence and nomenclature issues. In some cases, we simply want to replace an old implementation with an improved one. In others, we may need to maintain two implementations, one suitable for some clients, the other suitable for others. Indeed, a prime goal of this book is to consider in depth several implementations of each of a number of fundamental ADTs, generally with different performance characteristics. In this book, we often compare the performance of a single client using two

API

```
public class Date
          Date(int month, int day, int year)   create a date
   int    month()                              month
   int    day()                                day
   int    year()                               year
String    toString()                           string representation
```

test client

```
public static void main(String[] args)
{
   int m = Integer.parseInt(args[0]);
   int d = Integer.parseInt(args[1]);
   int y = Integer.parseInt(args[2]);
   Date date = new Date(m, d, y);
   StdOut.println(date);
}
```

application

```
% java Date 12 31 1999
12/31/1999
```

implementation

```
public class Date
{
   private final int month;
   private final int day;
   private final int year;

   public Date(int m, int d, int y)
   {  month = m; day = d; year = y; }

   public int month()
   {  return month;  }

   public int day()
   {  return day;  }

   public int year()
   {  return day;  }

   public String toString()
   {  return month() + "/" + day()
                     + "/" + year();  }

}
```

alternate implementation

```
public class Date
{
   private final int value;

   public Date(int m, int d, int y)
   { value = y*512 + m*32 + d; }

   public int month()
   { return (value / 32) % 16; }

   public int day()
   { return value % 32; }

   public int year()
   { return value / 512; }

   public String toString()
   {  return month() + "/" + day()
                     + "/" + year();  }
}
```

An abstract data type to encapsulate dates, with two implementations

different implementations of the same API. For this reason, we generally adopt an informal naming convention where we:

- Identify different implementations of the same API by prepending a descriptive modifier. For example, we might name our `Date` implementations on the previous page `BasicDate` and `SmallDate`, and we might wish to develop a `SmartDate` implementation that can validate that dates are legal.
- Maintain a reference implementation with no prefix that makes a choice that should be suitable for most clients. That is, most clients should just use `Date`.

In a large system, this solution is not ideal, as it might involve changing client code. For example, if we were to develop a new implementation `ExtraSmallDate`, then our only options are to change client code or to make it the reference implementation for use by all clients. Java has various advanced language mechanisms for maintaining multiple implementations without needing to change client code, but we use them sparingly because their use is challenging (and even controversial) even for experts, especially in conjuction with other advanced language features that we do value (generics and iterators). These issues are important (for example, ignoring them led to the celebrated *Y2K problem* at the turn of the millennium, because many programs used their own implementations of the date abstraction that did not take into account the first two digits of the year), but detailed consideration of these issues would take us rather far afield from the study of algorithms.

Accumulator. The *accumulator* API shown on the facing page defines an abstract data type that provides to clients the ability to maintain a running average of data values. For example, we use this data type frequently in this book to process experimental results (see SECTION 1.4). The implementation is straightforward: it maintains a `int` instance variable counts the number of data values seen so far and a `double` instance variable that keeps track of the sum of the values seen so far; to compute the average it divides the sum by the count. Note that the implementation does not save the data values—it could be used for a huge number of them (even on a device that is not capable of holding that many), or a huge number of accumulators could be used on a big system. This performance characteristic is subtle and might be specified in the API, because an implementation that does save the values might cause an application to run out of memory.

API

```
public class Accumulator
                 Accumulator()                 create an accumulator
           void  addDataValue(double val)      add a new data value
         double  mean()                        mean of all data values
         String  toString()                    string representation
```

typical client

```
public class TestAccumulator
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      Accumulator a = new Accumulator();
      for (int t = 0; t < T; t++)
         a.addDataValue(StdRandom.random());
      StdOut.println(a);
   }
}
```

application

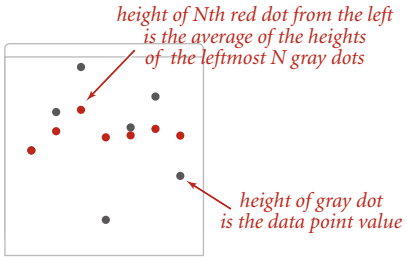
```
% java TestAccumulator 1000
Mean (1000 values): 0.51829

% java TestAccumulator 1000000
Mean (1000000 values): 0.49948

% java TestAccumulator 1000000
Mean (1000000 values): 0.50014
```

implementation

```
public class Accumulator
{
   private double total;
   private int N;

   public void addDataValue(double val)
   {
       N++;
       total += val;
   }

   public double mean()
   {  return total/N;  }

   public String toString()
   { return "Mean (" + N + " values): "
                + String.format("%7.5f", mean()); }

}
```

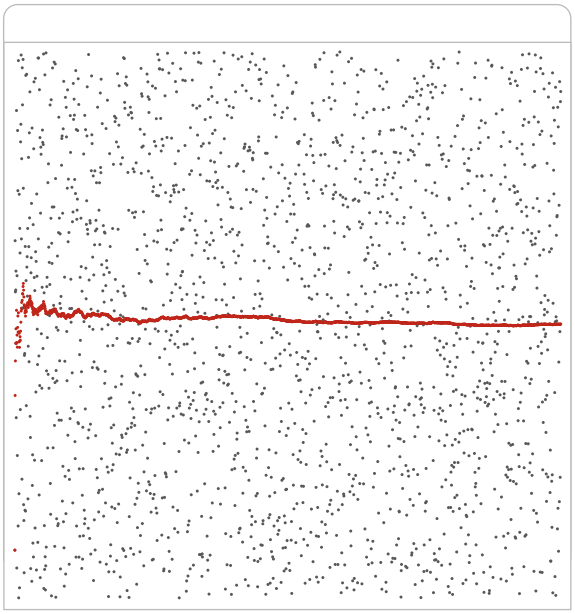
An abstract data type for accumulating data values

Visual accumulator. The *visual accumulator* implementation shown on the facing page extends `Accumulator` to present a useful side effect: it draws on `StdDraw` all the data (in gray) and the running average (in red). The easiest way to do so is to add a constructor that provides the number of points to be plotted and the maximum value, for rescaling the plot. `VisualAccumulator` is not technically an implementation of the `Accumulator` API (its constructor has a different signature and it causes a different prescribed side effect). Generally, we are careful to fully specify APIs and are loath to make *any* changes in an API once articulated, as it might involve changing an unknown amount of client (and implementation) code, but adding a constructor to gain functionality can sometimes be defended because it involves changing the same line in client code that we change when changing a class name. In this example, if we have developed a client that uses an `Accumulator` and perhaps has many calls to `addDataValue()` and `avg()`, we can enjoy the benefits of `VisualAccumulator` by just changing one line of client code.



Visual accumulator plot

application



```
% java TestVisualAccumulator 2000
Mean (2000 values): 0.509789
```

API

```
public class VisualAccumulator
             VisualAccumulator(int trials, double max)
       void  addDataValue(double val)    add a new data value
     double  avg()                       average of all data values
     String  toString()                  string representation
```

typical client

```
public class TestVisualAccumulator
{
   public static void main(String[] args)
   {
      int T = Integer.parseInt(args[0]);
      VisualAccumulator a = new VisualAccumulator(T, 1.0);
      for (int t = 0; t < T; t++)
         a.addDataValue(StdRandom.random());
      StdOut.println(a);
   }
}
```

implementation

```
public class VisualAccumulator
{
   private double total;
   private int N;

   public VisualAccumulator(int trials, double max)
   {
      StdDraw.setXscale(0, trials);
      StdDraw.setYscale(0, max);
      StdDraw.setPenRadius(.005);
   }

   public void addDataValue(double val)
   {
      N++;
      total += val;
      StdDraw.setPenColor(StdDraw.DARK_GRAY);
      StdDraw.point(N, val);
      StdDraw.setPenColor(StdDraw.RED);
      StdDraw.point(N, total/N);
   }

   public double mean()
   public String toString()
   // Same as Accumulator.

}
```

An abstract data type for accumulating data values (visual version)

Data-type design *An abstract data type is a data type whose representation is hidden from the client.* This idea has had a powerful effect on modern programming. The various examples that we have considered give us the vocabulary to address advanced characteristics of ADTs and their implementation as Java classes. Many of these topics are, on the surface, tangential to the study of algorithms, so it is safe for you to skim this section and refer to it later in the context of specific implementation problems. Our goal is to put important information related to designing data types in one place for reference and to set the stage for implementations throughout this book.

Encapsulation. A hallmark of object-oriented programming is that it enables us to *encapsulate* data types within their implementations, to facilitate separate development of clients and data type implementations. Encapsulation enables modular programming, allowing us to

- Independently develop of client and implementation code
- Substitute improved implementations without affecting clients
- Support programs not yet written (the API is a guide for any future client)

Encapsulation also isolates data-type operations, which leads to the possibility of

- Limiting the potential for error
- Adding consistency checks and other debugging tools in implementations
- Clarifying client code

An encapsulated data type can be used by any client, so it extends the Java language. The programming style that we are advocating is predicated on the idea of breaking large programs into small modules that can be developed and debugged independently. This approach improves the resiliency of our software by limiting and localizing the effects of making changes, and it promotes code reuse by making it possible to substitute new implementations of a data type to improve performance, accuracy, or memory footprint. The same idea works in many settings. We often reap the benefits of encapsulation when we use system libraries. New versions of the Java system often include new implementations of various data types or static method libraries, but *the APIs do not change*. In the context of the study of algorithms and data structures, there is strong and constant motivation to develop better algorithms because we can improve performance for *all* clients by substituting an improved ADT implementation without changing the code of *any* client. The key to success in modular programming is to maintain *independence* among modules. We do so by insisting on the API being the *only* point of dependence between client and implementation. *You do not need to know how a data type is implemented in order to use it* and *you can assume that a client knows nothing but the API* when implementing a data type. Encapsulation is the key to attaining both of these advantages.

Designing APIs. One of the most important and most challenging steps in building modern software is designing APIs. This task takes practice, careful deliberation, and many iterations, but any time spent designing a good API is certain to be repaid in time saved debugging or code reuse. Articulating an API might seem to be overkill when writing a small program, but you should consider writing *every* program as though you will need to reuse the code someday. Ideally, an API would clearly articulate behavior for all possible inputs, including side effects, and then we would have software to check that implementations meet the specification. Unfortunately, a fundamental result from theoretical computer science known as the *specification problem* implies that this goal is actually *impossible* to achieve. Briefly, such a specification would have to be written in a formal language like a programming language, and the problem of determining whether two programs perform the same computation is known, mathematically, to be *undecidable*. Therefore, our APIs are brief English-language descriptions of the set of values in the associated abstract data type along with a list of constructors and instance methods, again with brief English-language descriptions of their purpose, including side effects. To validate the design, we always include examples of client code in the text surrounding our APIs. Within this broad outline, there are numerous pitfalls that every API design is susceptible to:

- An API may be *too hard to implement*, implying implementations that are difficult or impossible to develop.
- An API may be *too hard to use*, leading to client code that is more complicated than it would be without the API.
- An API may be *too narrow*, omitting methods that clients need.
- An API may be *too wide*, including a large number of methods not needed by any client. This pitfall is perhaps the most common, and one of the most difficult to avoid. The size of an API tends to grow over time because it is not difficult to add methods to an existing API, but it *is* difficult to remove methods without breaking existing clients.
- An API may be *too general*, providing no useful abstractions.
- An API may be *too specific*, providing abstractions so detailed or so diffuse as to be useless.
- An API may be *too dependent on a particular representation*, therefore not serving the purpose of freeing client code from the details of using that representation. This pitfall is also difficult to avoid, because the representation is certainly central to the development of the implementation.

These considerations are sometimes summarized in yet another motto: *provide to clients the methods they need and no others.*

Algorithms and abstract data types. Data abstraction is naturally suited to the study of algorithms, because it helps us provide a framework within which we can precisely specify both what an algorithm needs to accomplish and how a client can make use of an algorithm. Typically, in this book, an algorithm is an implementation of an instance method in an abstract data type. For example, our whitelisting example at the beginning of the chapter is naturally cast as an ADT client, based on the following operations:

- Construct a `SET` from an array of given values.
- Determine whether a given value is in the set.

These operations are encapsulated in the `StaticSETofInts` ADT, shown on the facing page along with `Whitelist`, a typical client. `StaticSETofInts` is a special case of the more general and more useful *symbol table* ADT that is the focus of CHAPTER 3. Binary search is one of several algorithms that we study that is suitable for implementing these ADTs. By comparison with the `BinarySearch` implementation on page 47, this implementation leads to clearer and more useful client code. For example, `StaticSETofInts` enforces the idea that the array must be sorted before `rank()` is called. With the abstract data type, we separate the client from the implementation making it easier for *any* client to benefit from the ingenuity of the binary search algorithm, just by following the API (clients of `rank()` in `BinarySearch` have to know to sort the array first). Whitelisting is one of many clients that can take advantage of binary search.

EVERY JAVA PROGRAM is a set of static methods and/or a data type implementation. In this book, we focus primarily on *abstract* data type implementations such as `StaticSETofInts`, where the focus is on operations and the representation of the data is hidden from the client. As this example illustrates, data abstraction enables us to

application

```
% java Whitelist largeW.txt < largeT.txt
499569
984875
295754
207807
140925
161828
 ...
```

- Precisely specify what algorithms can provide for clients
- Separate algorithm implementations from the client code
- Develop layers of abstraction, where we make use of well-understood algorithms to develop other algorithms

These are desirable properties of *any* approach to describing algorithms, whether it be an English-language description or pseudo-code. By embracing the Java `class` mechanism in support of data abstraction, we have little to lose and much to gain: working code that we can test and use to compare performance for diverse clients.

API

```
public class StaticSETofInts
          StaticSETofInts(int[] a)     create a set from the values in a[]
boolean   contains(int key)            is key in the set?
```

typical client

```
public class Whitelist
{
   public static void main(String[] args)
   {
      int[] w = In.readInts(args[0]);
      StaticSETofInts set = new StaticSETofInts(w);
      while (!StdIn.isEmpty())
      {  // Read key, print if not in whitelist.
         int key = StdIn.readInt();
         if (set.rank(key) == -1)
            StdOut.println(key);
      }
   }
}
```

implementation

```
import java.util.Arrays;

public class StaticSETofInts
{
   private int[] a;

   public StaticSETofInts(int[] keys)
   {
      a = new int[keys.length];
      for (int i = 0; i < keys.length; i++)
         a[i] = keys[i]; // defensive copy
      Arrays.sort(a);
   }

   public boolean contains(int key)
   {  return rank(key) != -1;  }

   private int rank(int key)
   {  // Binary search.
      int lo  = 0;
      int hi = a.length - 1;
      while (lo <= hi)
      {  // Key is in a[lo..hi] or not present.
         int mid = lo + (hi - lo) / 2;
         if      (key < a[mid]) hi = mid - 1;
         else if (key > a[mid]) lo = mid + 1;
         else                   return mid;
      }
      return -1;
   }
}
```

Binary search recast as an object-oriented program (an ADT for search in a set of integers)

Interface inheritance. Java provides language support for defining relationships among objects, known as *inheritance*. These mechanisms are widely used by software developers, so you will study them in detail if you take a course in software engineering. The first inheritance mechanism that we consider is known as *subtyping*, which allows us to specify a relationship between otherwise unrelated classes by specifying in an *interface* a set of common methods that each implementing class must contain. An interface is nothing more than a list of instance methods. For example, instead of using our informal API, we might have articulated an interface for `Date`:

```
public interface Datable
{
   int month();
   int day();
   int year();
}
```

and then referred to the interface in our implementation code

```
public class Date implements Datable
{
   // implementation code (same as before)
}
```

so that the Java compiler will check that it matches the interface. Adding the code `implements Datable` to any class that implements `month()`, `day()`, and `year()` provides a guarantee to any client that an object of that class can invoke those methods. This arrangement is known as *interface inheritance*—an implementing class *inherits* the interface. Interface inheritance allows us to write client programs that can manipulate objects of *any* type that implements the interface (even a type to be created in the future), by invoking methods in the interface. We might have used interface inheritance in place of our more informal APIs, but chose not to do so to avoid dependence on specific high-level language mechanisms that are not critical to the understanding of algorithms and to avoid the extra baggage of interface files. But there are a few situations where Java conventions make

	interface	methods	section
comparison	`java.lang.Comparable`	`compareTo()`	2.1
	`java.util.Comparator`	`compare()`	2.5
iteration	`java.lang.Iterable`	`iterator()`	1.3

	`java.util.Iterator`	`hasNext()`
		`next()`
		`remove()`
			1.3

Java interfaces used in this book

it worthwhile for us to take advantage of interfaces: we use them for *comparison* and for *iteration*, as detailed in the table at the bottom of the previous page, and will consider them in more detail when we cover those concepts.

Implementation inheritance. Java also supports another inheritence mechanism known as *subclassing*, which is a powerful technique that enables a programmer to change behavior and add functionality without rewriting an entire class from scratch. The idea is to define a new class (*subclass*, or *derived class*) that inherits instance methods *and* instance variables from another class (*superclass*, or *base class*). The subclass contains more methods than the superclass. Moreover, the subclass can redefine or *override* methods in the superclass. Subclassing is widely used by systems programmers to build so-called *extensible* libraries—one programmer (even you) can add methods to a library built by another programmer (or, perhaps, a team of systems programmers), effectively reusing the code in a potentially huge library. For example, this approach is widely used in the development of graphical user interfaces, so that the large amount of code required to provide all the facilities that users expect (drop-down menus, cut-and-paste, access to files, and so forth) can be reused. The use of subclassing is controversial among systems and applications programmers (its advantages over interface inheritance are debatable), and we avoid it in this book because it generally works against encapsulation. Certain vestiges of the approach are built in to Java and therefore unavoidable: specifically, every class is a subtype of Java's `Object` class. This structure enables the "convention" that every class includes an implementation of `getClass()`, `toString()`, `equals()`, `hashCode()`, and several other methods that we do not use in this book. Actually, every class *inherits* these methods from `Object` through subclassing, so any client can use them for any object. We usually override `toString()`, `equals()`, `hashCode()` in new classes because the default `Object` implementation generally does not lead to the desired behavior. We now will consider `toString()` and `equals()`; we discuss `hashCode()` in SECTION 3.4.

	method	purpose	section
`Class`	`getClass()`	*what class is this object?*	1.2
`String`	`toString()`	*string representation of this object*	1.1
`boolean`	`equals(Object that)`	*is this object equal to* `that`*?*	1.2
`int`	`hashCode()`	*hash code for this object*	3.4

Inherited methods from `Object` used in this book

String conversion. By convention, every Java type inherits `toString()` from `Object`, so any client can invoke `toString()` for any object. This convention is the basis for Java's automatic conversion of one operand of the concatenation operator + to a `String` whenever the other operand is a `String`. If an object's data type does not include an implementation of `toString()`, then the default implementation in `Object` is invoked, which is normally not helpful, since it typically returns a string representation of the memory address of the object. Accordingly, we generally include implementations of `toString()` that override the default in every class that we develop, as highlighted for `Date` on the facing page. As illustrated in this code, `toString()` implementations are often quite simple, implicitly (through +) using `toString()` for each instance variable.

Wrapper types. Java supplies built-in reference types known as *wrapper types*, one for each of the primitive types: `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, and `Short` correspond to `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short`, respectively. These classes consist primarily of static methods such as `parseInt()` but they also include the inherited instance methods `toString()`, `compareTo()`, `equals()`, and `hashCode()`. Java automatically converts from primitive types to wrapper types when warranted, as described on page 122. For example, when an `int` value is concatenated with a `String`, it is converted to an `Integer` that can invoke `toString()`.

Equality. What does it mean for two objects to be equal? If we test equality with `(a == b)` where `a` and `b` are reference variables of the same type, we are testing whether they have the same identity: whether the *references* are equal. Typical clients would rather be able to test whether the *data-type values* (object state) are the same, or to implement some type-specific rule. Java gives us a head start by providing implementations both for standard types such as `Integer`, `Double`, and `String` and for more complicated types such as `File` and `URL`. When using these types of data, you can just use the built-in implementation. For example, if `x` and `y` are `String` values, then `x.equals(y)` is `true` if and only if `x` and `y` have the same length and are identical in each character position. When we define our own data types, such as `Date` or `Transaction`, we need to override `equals()`. Java's convention is that `equals()` must be an *equivalence relation*. It must be

- *Reflexive*: `x.equals(x)` is `true`.
- *Symmetric*: `x.equals(y)` is `true` if and only if `y.equals(x)`.
- *Transitive*: if `x.equals(y)` and `y.equals(z)` are `true`, then so is `x.equals(z)`.

In addition, it must take an `Object` as argument and satisfy the following properties.

- *Consistent*: multiple invocations of `x.equals(y)` consistently return the same value, provided neither object is modified.
- *Not null*: `x.equals(null)` returns `false`.

These are natural definitions, but ensuring that these properties hold, adhering to Java conventions, and avoiding unnecessary work in an implementation can be tricky, as illustrated for `Date` below. It takes the following step-by-step approach:

- If the reference to this object is the same as the reference to the argument object, return `true`. This test saves the work of doing all the other checks in this case.
- If the argument is null, return `false`, to adhere to the convention (and to avoid following a null reference in code to follow).
- If the objects are not from the same class, return `false`. To determine an object's class, we use `getClass()`. Note that we can use == to tell us whether two objects of type `Class` are equal because `getClass()` is guaranteed to return the same reference for all objects in any given class.
- Cast the argument from `Object` to `Date` (this cast must succeed because of the previous test).
- Return `false` if any instance variables do not match. For other classes, some other definition of equality might be appropriate. For example, we might regard two `Counter` objects as equal if their `count` instance variables are equal.

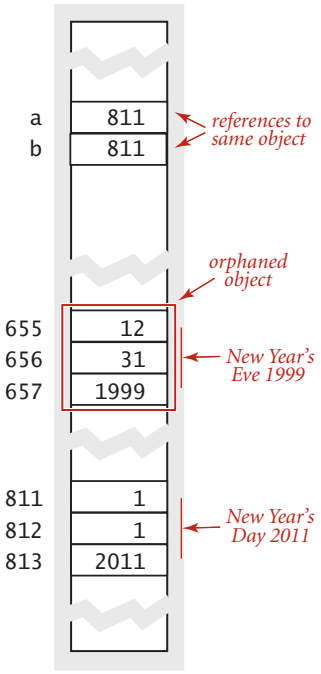
This implementation is a model that you can use to implement `equals()` for any type that you implement. Once you have implemented one `equals()`, you will not find it difficult to implement another.

```
public class Date
{
   private final int month;
   private final int day;
   private final int year;

   public Date(int m, int d, int y)
   {  month = m; day = d; year = y; }

   public int month()
   {  return month;  }

   public int day()
   {  return day;  }

   public int year()
   {  return year;  }

   public String toString()
   {  return month() + "/" + day() + "/" + year();  }

   public boolean equals(Object x)
   {
      if (this == x) return true;
      if (x == null) return false;
      if (this.getClass() != x.getClass()) return false;
      Date that = (Date) x;
      if (this.day != that.day)            return false;
      if (this.month != that.month)        return false;
      if (this.year != that.year)          return false;
      return true;
   }

}
```

Overriding `toString()` and `equals()` in a data-type definition

Memory management. The ability to assign a new value to a reference variable creates the possibility that a program may have created an object that can no longer be referenced. For example, consider the three assignment statements in the figure at left. After the third assignment statement, not only do `a` and `b` refer to the same `Date` object (1/1/2011), but also there is no longer a reference to the `Date` object that was created and used to initialize `b`. The only reference to that object was in the variable `b`, and this reference was overwritten by the assignment, so there is no way to refer to the object again. Such an object is said to be *orphaned*. Objects are also orphaned when they go out of scope. Java programs tend to create huge numbers of objects (and variables that hold primitive data-type values), but only have a need for a small number of them at any given point in time. Accordingly, programming languages and systems need mechanisms to *allocate* memory for data-type values during the time they are needed and to *free* the memory when they are no longer needed (for an object, sometime after it is orphaned). Memory management turns out to be easier for primitive types because all of the information needed for memory allocation is known at compile time. Java (and most other systems) takes care of reserving space for variables when they are declared and freeing that space when they go out of scope. Memory management for objects is more complicated: the system can allocate memory for an object when it is created, but cannot know precisely when to free the memory associated with each object because the dynamics of a program in execution determines when objects are orphaned. In many languages (such as C and C++) the programmer is responsible for both allocating and freeing memory. Doing so is tedious and notoriously error-prone. One of Java's most significant features is its ability to *automatically* manage memory. The idea is to free the programmers from the responsibility of managing memory by keeping track of orphaned objects and returning the memory they use to a pool of free memory. Reclaiming memory in this way is known as *garbage collection*. One of Java's characteristic features is its policy that references cannot be modified. This policy enables Java to do efficient automatic garbage collection. Programmers still debate whether the overhead of automatic garbage collection justifies the convenience of not having to worry about memory management.

```
Date a = new Date(12, 31, 1999);
Date b = new Date(1, 1, 2011);
b = a;
```



An orphaned object

Immutability. An *immutable* data type, such as `Date`, has the property that the value of an object never changes once constructed. By contrast, a *mutable* data type, such as `Counter` or `Accumulator`, manipulates object values that are intended to change. Java's language support for helping to enforce immutability is the `final` modifier. When you declare a variable to be `final`, you are promising to assign it a value only once, either in an initializer or in the constructor. Code that could modify the value of a `final` variable leads to a compile-time error. In our code, we use the modifier `final` with instance variables whose values never change. This policy serves as documentation that the value does not change, prevents accidental changes, and makes programs easier to debug. For example, you do not have to include a `final` value in a trace, since you know that its value never changes. A data type such as `Date` whose instance variables are all primitive and `final` is immutable (in code that does not use implementation inheritence, our convention). Whether to make a data type immutable is an important design decision and depends on the application at hand. For data types such as `Date`, the purpose of the abstraction is to encapsulate values that do not change so that we can use them in assignment statements and as arguments and return values from functions in the same way as we use primitive types (without having to worry about their values changing). A programmer implementing a `Date` client might reasonably expect to write the code `d = d0` for two `Date` variables, in the same way as for `double` or `int` values. But if `Date` were mutable and the value of `d` were to change *after* the assignment `d = d0`, then the value of `d0` would *also* change (they are both references to the same object)! On the other hand, for data types such as `Counter` and `Accumulator`, the very purpose of the abstraction is to encapsulate values as they change. You have already encountered this distinction as a client programmer, when using Java arrays (mutable) and Java's `String` data type (immutable). When you pass a `String` to a method, you do not worry about that method changing the sequence of characters in the `String`, but when you pass an array to a method, the method is free to change the contents of the array. `String` objects are immutable because we generally do *not* want `String` values to change, and Java arrays are mutable because we generally *do* want array values to change. There are also situations where we want to have mutable strings (that is the purpose of Java's `StringBuilder` class) and where we want to have immutable arrays (that is the purpose of the `Vector` class that we consider later in this section). Generally, immutable types are easier to use and harder to misuse than mutable types because the scope of code that can change their values is far smaller. It is easier to debug code that uses immutable types because it is easier to guarantee that variables in client code that uses them remain in a consistent state. When using mutable types,

mutable	immutable
`Counter`	`Date`
Java arrays	`String`

Mutable/immutable examples

you must always be concerned about where and when their values change. The downside of immutability is that *a new object must be created for every value*. This expense is normally manageable because Java garbage collectors are typically optimized for such situations. Another downside of immutability stems from the fact that, unfortunately, `final` guarantees immutability only when instance variables are primitive types, not reference types. If an instance variable of a reference type has the `final` modifier, the value of that instance variable (the reference to an object) will never change—it will always refer to the same object—but the value of the object itself *can* change. For example, this code does *not* implement an immutable type:

```
public class Vector
{
   private final double[] coords;

   public Vector(double[] a)
   {  coords = a; }
   ...
}
```

A client program could create a `Vector` by specifying the entries in an array, and then (bypassing the API) change the elements of the `Vector` after construction:

```
double[] a = { 3.0, 4.0 };
Vector vector = new Vector(a);
a[0] = 0.0;  // Bypasses the public API.
```

The instance variable `coords[]` is `private` and `final`, but `Vector` is mutable because the client holds a reference to the data. Immutability needs to be taken into account in any data-type design, and whether a data type is immutable should be specified in the API, so that clients know that object values will not change. In this book, our primary interest in immutability is for use in certifying the correctness of our algorithms. For example, if the type of data used for a binary search algorithm were mutable, then clients could invalidate our assumption that the array is sorted for binary search.

Design by contract. To conclude, we briefly discuss Java language mechanisms that enables you to verify assumptions about your program *as it is running*. We use two Java language mechanisms for this purpose:

- Exceptions, which generally handle unforeseen errors *outside* our control
- Assertions, which verify assumptions that we make *within* code we develop

Liberal use of both exceptions and assertions is good programming practice. We use them sparingly in the book for economy, but you will find them throughout the code on the booksite. This code aligns with a substantial amount of the surrounding commentary about each algorithm in the text that has to do with exceptional conditions and with asserted invariants.

Exceptions and errors. *Exceptions* and *errors* are disruptive events that occur while a program is running, often to signal an error. The action taken is known as *throwing an exception or throwing an error*. We have already encountered exceptions thrown by Java system methods in the course of learning basic features of Java: `StackOverflowError`, `ArithmeticException`, `ArrayIndexOutOfBoundsException`, `OutOfMemoryError`, and `NullPointerException` are typical examples. You can also create your own exceptions. The simplest kind is a `RuntimeException` that terminates execution of the program and prints an error message

```
throw new RuntimeException("Error message here.");
```

A general practice known as *fail fast* programming suggests that an error is more easily pinpointed if an exception is thrown as soon as an error is discovered (as opposed to ignoring the error and deferring the exception to sometime in the future).

Assertions. An *assertion* is a boolean expression that you are affirming is `true` at that point in the program. If the expression is `false`, the program will terminate and report an error message. We use assertions both to gain confidence in the correctness of programs and to document intent. For example, suppose that you have a computed value that you might use to index into an array. If this value were negative, it would cause an `ArrayIndexOutOfBoundsException` sometime later. But if you write the code `assert index >= 0;` you can pinpoint the place where the error occurred. You can also add an optional detail message such as

```
assert index >= 0 : "Negative index in method X";
```

to help you locate the bug. By default, assertions are disabled. You can enable them from the command line by using the `-enableassertions` flag (`-ea` for short). Assertions are for debugging: your program should not rely on assertions for normal operation since they may be disabled. When you take a course in systems programming, you will learn to use assertions to ensure that your code *never* terminates in a system error or goes into

an infinite loop. One model, known as the *design-by-contract* model of programming expresses the idea. The designer of a data type expresses a *precondition* (the condition that the client promises to satisfy when calling a method), a *postcondition* (the condition that the implementation promises to achieve when returning from a method), and *side effects* (any other change in state that the method could cause). During development, these conditions can be tested with assertions.

Summary. The language mechanisms discussed throughout this section illustrate that effective data-type design leads to nontrivial issues that are not easy to resolve. Experts are still debating the best ways to support some of the design ideas that we are discussing. Why does Java not allow functions as arguments? Why does Matlab copy arrays passed as arguments to functions? As mentioned early in CHAPTER 1, it is a slippery slope from complaining about features in a programming language to becoming a programming-language designer. If you do not plan to do so, your best strategy is to use widely available languages. Most systems have extensive libraries that you certainly should use when appropriate, but you often can simplify your client code and protect yourself by building abstractions that can easily transport to other languages. Your main goal is to develop data types so that most of your work is done at a level of abstraction that is appropriate to the problem at hand.

The table on the facing page summarizes the various kinds of Java classes that we have considered.

kind of class	examples	characteristics
static methods	`Math StdIn StdOut`	no instance variables
immutable abstract data type	`Date Transaction String Integer`	instance variables all `private` instance variables all `final` defensive copy for reference types *Note: these are necessary but not sufficient.*
mutable abstract data type	`Counter Accumulator`	instance variables all `private` not all instance variables `final`
abstract data type with I/O side effects	`VisualAccumulator In Out Draw`	instance variables all `private` instance methods do I/O

Java classes (data-type implementations)

Q & A

Q. Why bother with data abstraction?

A. It helps us produce reliable and correct code. For example, in the 2000 presidential election, Al Gore received –16,022 votes on an electronic voting machine in Volusia County, Florida—the tally was clearly not properly encapsulated in the voting machine software!

Q. Why the distinction between primitive and reference types? Why not just have reference types?

A. Performance. Java provides the reference types `Integer`, `Double`, and so forth that correspond to primitive types that can be used by programmers who prefer to ignore the distinction. Primitive types are closer to the types of data that are supported by computer hardware, so programs that use them usually run faster than programs that use corresponding reference types.

Q. Do data types *have* to be abstract?

A. No. Java also allows `public` and `protected` to allow some clients to refer directly to instance variables. As described in the text, the advantages of allowing client code to directly refer to data are greatly outweighed by the disadvantages of dependence on a particular representation, so all instance variables are `private` in our code. We also occasionally use `private` instance methods to share code among public methods.

Q. What happens if I forget to use `new` when creating an object?

A. To Java, it looks as though you want to call a static method with a return value of the object type. Since you have not defined such a method, the error message is the same as anytime you refer to an undefined symbol. If you compile the code

```
Counter c = Counter("test");
```

you get this error message:

```
cannot find symbol
symbol  : method Counter(String)
```

You get the same kind of error message if you provide the wrong number of arguments to a constructor.

Q. What happens if I forget to use `new` when creating an array of objects?

A. You need to use `new` for each object that you create, so when you create an array of N objects, you need to use `new` N+1 times: once for the array and once for each of the objects. If you forget to create the array:

```
Counter[] a;
a[0] = new Counter("test");
```

you get the same error message that you would get when trying to assign a value to any uninitialized variable:

```
variable a might not have been initialized
      a[0] = new Counter("test");
      ^
```

but if you forget to use `new` when creating an object within the array and then try to use it to invoke a method:

```
Counter[] a = new Counter[2];
a[0].increment();
```

you get a `NullPointerException`.

Q. Why not write `StdOut.println(x.toString())` to print objects?

A. That code works fine, but Java saves us the trouble of writing it by automatically invoking the `toString()` method for any object, since `println()` has a method that takes an `Object` as argument.

Q. What is a *pointer*?

A. Good question. Perhaps that should be `NullReferenceException`. Like a Java reference, you can think of a *pointer* as a machine address. In many programming languages, the pointer is a primitive data type that programmers can manipulate in many ways. But programming with pointers is notoriously error-prone, so operations provided for pointers need to be carefully designed to help programmers avoid errors. Java takes this point of view to an extreme (that is favored by many modern programming-language designers). In Java, there is only *one* way to create a reference (`new`) and only *one* way to change a reference (with an assignment statement). That is, the only things that a programmer can do with references are to create them and copy them. In

Q & A *(continued)*

programming-language jargon, Java references are known as *safe pointers*, because Java can guarantee that each reference points to an object of the specified type (and it can determine which objects are not in use, for garbage collection). Programmers used to writing code that directly manipulates pointers think of Java as having no pointers at all, but people still debate whether it is really desirable to have unsafe pointers.

Q. Where can I find more details on how Java implements references and does garbage collection?

A. One Java system might differ completely from another. For example, one natural scheme is to use a pointer (machine address); another is to use a *handle* (a pointer to a pointer). The former gives faster access to data; the latter provides for better garbage collection.

Q. What exactly does it mean to `import` a name?

A. Not much: it just saves some typing. You could type `java.util.Arrays` instead of `Arrays` everywhere in your code instead of using the `import` statement.

Q. What is the problem with implementation inheritance?

A. Subtyping makes modular programming more difficult for two reasons. First, any change in the superclass affects all subclasses. The subclass cannot be developed *independently* of the superclass; indeed, it is *completely dependent* on the superclass. This problem is known as the *fragile base class* problem. Second, the subclass code, having access to instance variables, can subvert the intention of the superclass code. For example, the designer of a class like `Counter` for a voting system may take great care to make it so that `Counter` can only increment the tally by one (remember Al Gore's problem). But a subclass, with full access to the instance variable, can change it to any value whatever.

Q. How do I make a class immutable?

A. To ensure immutability of a data type that includes an instance variable of a mutable type, we need to make a local copy, known as a *defensive copy*. And that may not be enough. Making the copy is one challenge; ensuring that none of the instance methods change values is another.

Q. What is `null`?

A. It is a literal value that refers to no object. Invoking a method using the `null` reference is meaningless and results in a `NullPointerException`. If you get this error message, check to make sure that your constructor properly initializes all of its instance variables.

Q. Can I have a static method in a class that implements a data type?

A. Of course. For example, all of our classes have `main()`. Also, it is natural to consider adding static methods for operations that involve multiple objects where none of them naturally suggests itself as the one that should invoke the method. For example, we might define a static method like the following within `Point`:

```
public static double distance(Point a, Point b)
{
   return a.distTo(b);
}
```

Often, including such methods can serve to clarify client code.

Q. Are there other kinds of variables besides parameter, local, and instance variables?

A. If you include the keyword `static` in a class declaration (outside of any type) it creates a completely different type of variable, known as a *static variable*. Like instance variables, static variables are accessible to every method in the class; however, they are not associated with any object. In older programming languages, such variables are known as *global variables*, because of their global scope. In modern programming, we focus on limiting scope and therefore rarely use such variables. When we do, we will call attention to them.

Q. What is a *deprecated* method?

A. A method that is no longer fully supported, but kept in an API to maintain compatibility. For example, Java once included a method `Character.isSpace()`, and programmers wrote programs that relied on using that method's behavior. When the designers of Java later wanted to support additional Unicode whitespace characters, they could not change the behavior of `isSpace()` without breaking client programs, so, instead, they added a new method, `Character.isWhiteSpace()`, and deprecated the old method. As time wears on, this practice certainly complicates APIs. Sometimes, entire classes are deprecated. For example, Java deprecated its `java.util.Date` in order to better support internationalization.

EXERCISES

1.2.1 Write a `Point2D` client that takes an integer value *N* from the command line, generates *N* random points in the unit square, and computes the distance separating the *closest pair* of points.

1.2.2 Write an `Interval1D` client that takes an `int` value *N* as command-line argument, reads *N* intervals (each defined by a pair of `double` values) from standard input, and prints all pairs that intersect.

1.2.3 Write an `Interval2D` client that takes command-line arguments `N`, `min`, and `max` and generates `N` random 2D intervals whose width and height are uniformly distributed between `min` and `max` in the unit square. Draw them on `StdDraw` and print the number of pairs of intervals that intersect and the number of intervals that are contained in one another.

1.2.4 What does the following code fragment print?

```
String string1 = "hello";
String string2 = string1;
string1 = "world";
StdOut.println(string1);
StdOut.println(string2);
```

1.2.5 What does the following code fragment print?

```
String s = "Hello World";
s.toUpperCase();
s.substring(6, 11);
StdOut.println(s);
```

Answer: `"Hello  World"`. `String` objects are immutable—string methods return a new `String` object with the appropriate value (but they do not change the value of the object that was used to invoke them). This code ignores the objects returned and just prints the original string. To print `"WORLD"`, use `s = s.toUpperCase()` and `s = s.substring(6, 11)`.

1.2.6 A string `s` is a *circular rotation* of a string `t` if it matches when the characters are circularly shifted by any number of positions; e.g., `ACTGACG` is a circular shift of `TGACGAC`, and vice versa. Detecting this condition is important in the study of genomic sequences. Write a program that checks whether two given strings `s` and `t` are circular

shifts of one another. *Hint*: The solution is a one-liner with `indexOf()`, `length()`, and string concatenation.

1.2.7 What does the following recursive function return?

```
public static String mystery(String s)
{
   int N = s.length();
   if (N <= 1) return s;
   String a = s.substring(0, N/2);
   String b = s.substring(N/2, N);
   return mystery(b) + mystery(a);
}
```

1.2.8 Suppose that `a[]` and `b[]` are each integer arrays consisting of millions of integers. What does the follow code do? Is it reasonably efficient?

```
int[] t = a; a = b; b = t;
```

Answer. It swaps them. It could hardly be more efficient because it does so by copying references, so that it is not necessary to copy millions of elements.

1.2.9 Instrument `BinarySearch` (page 47) to use a `Counter` to count the total number of keys examined during all searches and then print the total after all searches are complete. *Hint*: Create a `Counter` in `main()` and pass it as an argument to `rank()`.

1.2.10 Develop a class `VisualCounter` that allows both increment and decrement operations. Take two arguments `N` and `max` in the constructor, where `N` specifies the maximum number of operations and `max` specifies the maximum absolute value for the counter. As a side effect, create a plot showing the value of the counter each time its tally changes.

1.2.11 Develop an implementation `SmartDate` of our `Date` API that raises an exception if the date is not legal.

1.2.12 Add a method `dayOfTheWeek()` to `SmartDate` that returns a `String` value `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, `Saturday`, or `Sunday`, giving the appropriate day of the week for the date. You may assume that the date is in the 21st century.

EXERCISES *(continued)*

1.2.13 Using our implementation of `Date` as a model (page 91), develop an implementation of `Transaction`.

1.2.14 Using our implementation of `equals()` in `Date` as a model (page 103), develop implementations of `equals()` for `Transaction`.

CREATIVE PROBLEMS

1.2.15 *File input.* Develop a possible implementation of the static `readInts()` method from `In` (which we use for various test clients, such as binary search on page 47) that is based on the `split()` method in `String`.

Solution:

```
public static int[] readInts(String name)
{
   In in = new In(name);
   String input = StdIn.readAll();
    String[] words = input.split("\\s+");
   int[] ints = new int[words.length;
   for int i = 0; i < word.length; i++)
      ints[i] = Integer.parseInt(words[i]);
   return ints;
}
```

We will consider a different implementation in SECTION 1.3 (see page 126).

1.2.16 *Rational numbers.* Implement an immutable data type `Rational` for rational numbers that supports addition, subtraction, multiplication, and division.

public class Rational		
	Rational(int numerator. int denominator)	
Rational	plus(Rational b)	*sum of this number and b*
Rational	minus(Rational b)	*difference of this number and b*
Rational	times(Rational b)	*product of this number and b*
Rational	divides(Rational b)	*quotient of this number and b*
boolean	equals(Rational that)	*is this number equal to* `that` *?*
String	toString()	*string representation*

You do not have to worry about testing for overflow (see EXERCISE 1.2.17), but use as instance variables two `long` values that represent the numerator and denominator to limit the possibility of overflow. Use Euclid's algorithm (see page 4) to ensure that the numerator and denominator never have any common factors. Include a test client that exercises all of your methods.

CREATIVE PROBLEMS *(continued)*

1.2.17 *Robust implementation of rational numbers.* Use assertions to develop an implementation of `Rational` (see EXERCISE 1.2.16) that is immune to overflow.

1.2.18 *Variance for accumulator.* Validate that the following code, which adds the methods `var()` and `stddev()` to `Accumulator`, computes both the mean and variance of the numbers presented as arguments to `addDataValue()`:

```
public class Accumulator
{
   private double m;
   private double s;
   private int N;

   public void addDataValue(double x)
   {
      N++;
      s = s + 1.0 * (N-1) / N * (x - m) * (x - m);
      m = m + (x - m) / N;
   }

   public double mean()
   {  return m;  }

   public double var()
   {  return s/(N - 1);  }

   public double stddev()
   {  return Math.sqrt(this.var());  }

}
```

This implementation is less susceptible to roundoff error than the straightforward implementation based on saving the sum of the squares of the numbers.

1.2.19 *Parsing.* Develop the parse constructors for your `Date` and `Transaction` implementations of EXERCISE 1.2.13 that take a single `String` argument to specify the initialization values, using the formats given in the table below.

Partial solution:

```
public Date(String date)
{
   String[] fields = date.split("/");
   month = Integer.parseInt(fields[0]);
   day   = Integer.parseInt(fields[1]);
   year  = Integer.parseInt(fields[2]);
}
```

type	format	example
`Date`	integers separated by slashes	`5/22/1939`
`Transaction`	customer, date, and amount, separated by whitespace	`Turing 5/22/1939 11.99`

Formats for parsing