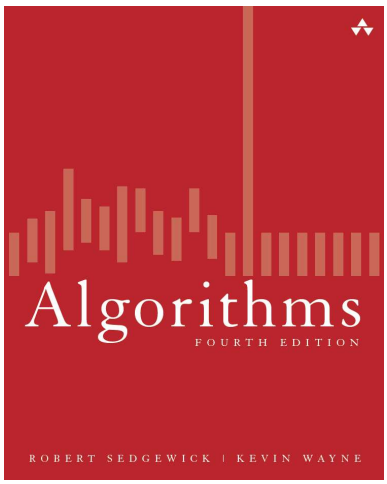
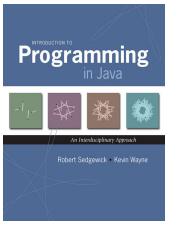
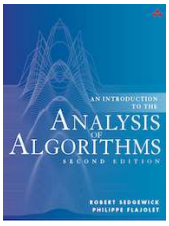






- Web Resources
- FAQ
- Data
- Code
- Errata
- Cheatsheet
- References
- Online Course
- Lecture Slides
- Programming Assignments

Sear ch

1.2 Data Abstraction

Object-oriented programming.

Programming in Java is largely based on building data types. 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. While 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, or 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 range ofdata types available in Java programming is open-ended, because you can define your own data types.

- *Data types.* A *data type* is a set of values and a set of operations on those values.
- *Abstract data types.* An *abstract data type* is a data type whose internal representation is hidden from the client.
- *Objects.* An *object* is an entity that can take on a data-type value. Objects are characterized by three essential properties. The *state* of an object is a value from its data type; the *identity* of an object distinguishes one object from another; the *behavior* of an object is the effect of data-type operations. In Java, a *reference* is a mechanism for accessing an object.
- *Applications programming interface (API).* 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
```

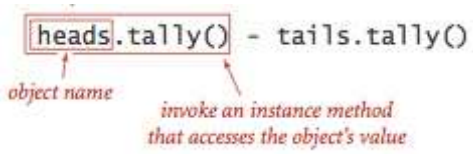
- *Client.* A client is a program that uses a data type.
- *Implementation.* An implementation is the code that implements the data type specified in an API.

Using abstract data types.

A client does not need to know how a data type is implemented in order to be able to use it.

- *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`. Each time that a client uses `new`, the system allocates memory space for the object, initializes its value, and returns a reference to the object.

- *Invoking instance methods.* The purpose of an instance method is to operate on data-type values. Instance methods have all of the properties of static methods: arguments are passed by value, method names can be overloaded, they may have a return value, and they may cause side effects. They have an additional property that characterizes them: *each invocation is associated with an object*.



- *Using objects.* Declarations give us variable names for objects that we can use in code. To use a given data type, we:
  - 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, Flips.java is a Counter.java client that takes a command-line argument T and simulates T coin flips.

- *Assignment statements.* An assignment statement with a reference type creates a copy of the reference (a does not create a new object). This situation is known as *aliasing*: both variables refer to the same object. 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);
```

The code prints the string `"2 ones"`.

- *Objects as arguments.* You can pass objects as arguments to methods. Java passes a *copy* of the argument value from the calling program to the method. This arrangement is known as *pass by value*. If you pass a reference to an object of type `Counter`, Java passes a copy of that reference. Thus, 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.* 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 FlipsMax.java, 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.

- *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

- *Arrays of objects.* Array entries can be of any type. 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. Rolls.java simulates rolling a die, using an array of `Counter` objects to keep track of the number of occurrences of each possible value.

Examples of abstract data types.

- *Geometric objects.* A natural example of object-oriented programming is designing data types for geometric objects.
  - Point2D.java is a data type for points in the plane.
  - Interval1D.java is a data type for one-dimensional intervals.
  - Interval2D.java is a data type for two-dimensional intervals.
- *Information processing.* Abstract data types provide a natural mechanism for organizing and processing information. the information
  - Date.java is a data type that represents the day, month, and year.
  - Transaction.java is a data type that represents a customer, a date, and an amount.
- *Accumulator.* Accumulator.java defines an ADT 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. VisualAccumulator.java in an enhanced version that also plots the data (in gray) and the running average (in red).

- *Strings.* Java's `String` data type in 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*-1st 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` 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.

- *Input and output revisited.* A disadvantage of the `StdIn`, `StdOut`, and `StdDraw` 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 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, your standard library includes the data types In.java, Out.java, and Draw.java that support multiple input and output streams.

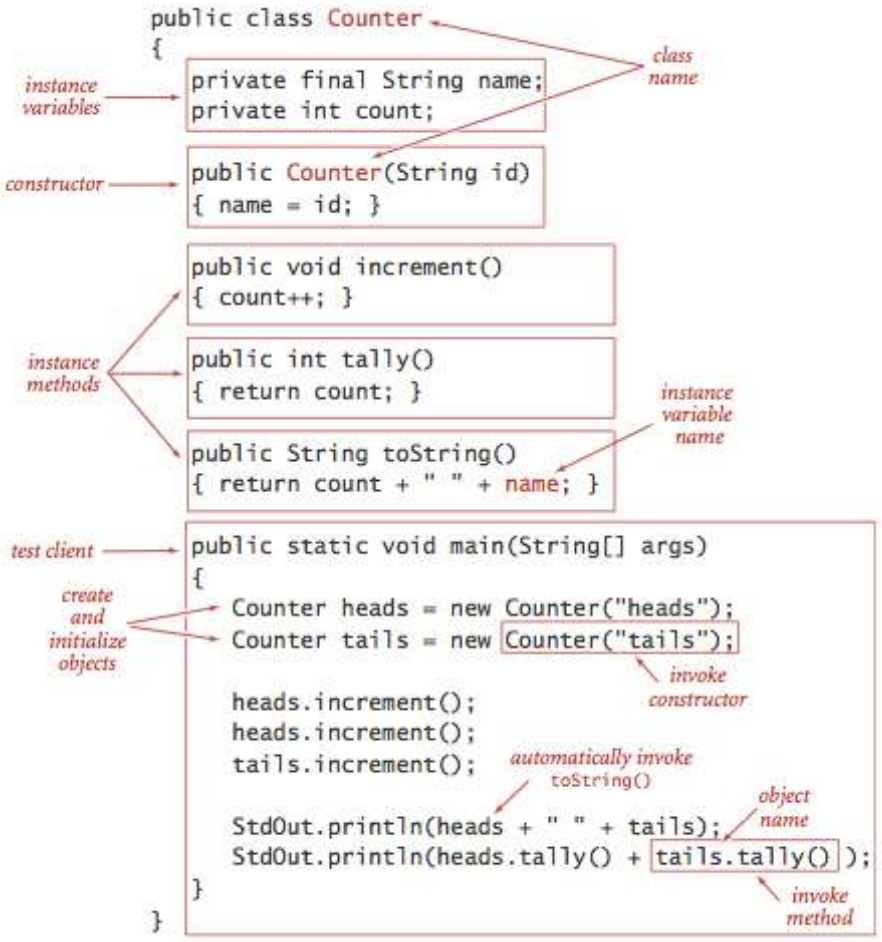
Implementing abstract data types.

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.

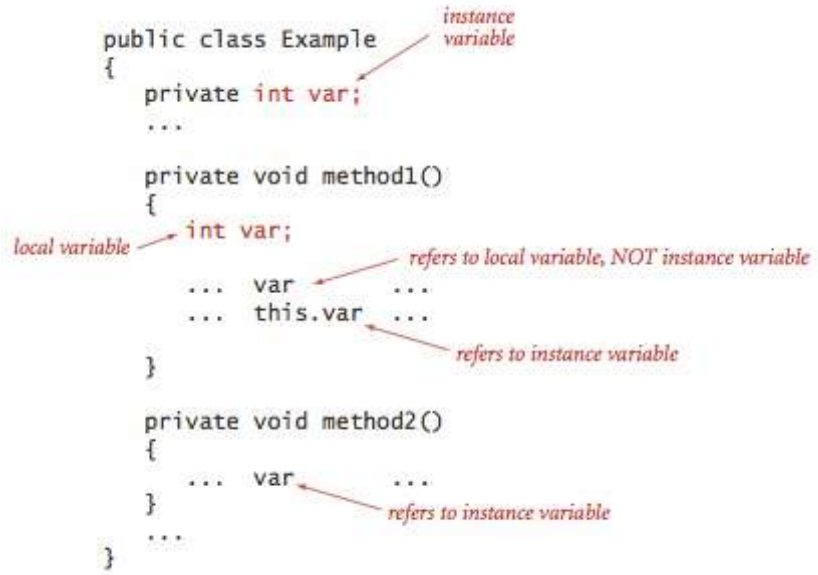
- *Instance variables.* To define data-type values (the state of each object), we declare instance variables in much the same way as we declare local variables. There are numerous values corresponding to each instance variable (one for each object that is an instance of the data type). Each declaration is qualified by a *visibility modifier*. In ADT implementations, we use `private`, using a Java language mechanism 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.

- *Constructors.* The constructor establishes an object's identity and initializes the instance variables. 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 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`.

- *Instance methods.* Instance methods specify the data-type operations. 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 those values, the statements are executed until a return value is computed, and the value returned to the client. Instance methods may be *public* (specified in the API) or *private* (used to organize the computation and not available to clients).



- *Scope.* Instance methods use three kinds of variables: parameter variables, local variables, and 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 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).



Designing abstract data types.

We 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.

- *Designing APIs.* One of the most important and most challenging steps in building modern software is designing APIs. 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. There are numerous potential pitfalls when designing an API:

  - Too hard to implement, making it difficult or impossible to develop.
  - Too hard to use, leading to complicated client code.
  - Too narrow, omitting methods that clients need.
  - Too wide, including a large number of methods not needed by any client.
  - Too general, providing no useful abstractions.
  - Too specific, providing an abstraction so rigid as to be useless.
  - Too dependent on a particular representation, therefore not freeing client code from the details of the representation.

  In summary, provide to clients the methods they need and no others.

- *Algorithms and ADTs.* 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. 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 StaticSETofInts.java and Whitelist.java.

- *Interface inheritance.* Java provides language support for defining relationships among objects, known as *inheritance*. The first inheritance mechanism that we consider is known as *subtyping*, which allows us to specify a relationship between otherwise unrelated classes by specifying an *interface* a set of common methods that each implementing class must contain. We use interface inheritance for *comparison* and for *iteration*.

	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

- *Implementation inheritance.* Java also supports another inheritance mechanism known as *subclassing*, which is a powerful technique that enables a programmer to change behavior and add functionality witho rewriting an entire class from scratch. The idea is to define a new class (*subclass*) that inherits instance methods and instance variables from another class (*superclass*). We avoid subclassing in this book because i generally works against encapsulation. Certain vestiges of the approach are built in to Java and therefore unavoidable—specifically, every class is a subclass of Object.

	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

- *String conversion.* Every Java type inherits `toString()` from 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`. We generally include implementations of `toString()` that override the default, as in Date.java and Transaction.java.

- *Wrapper types.* Java supplies built-in reference types known as *wrapper types*, one for each of the primitive types:

primitive type	wrapper type
boolean	Boolean
byte	Byte
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

  Java automatically converts from primitive types to wrapper types (*autoboxing*) and back (*auto-unboxing*) when warranted.

- *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. Every Java type inherits the method `equals()` from Object. Java provides natural implementations both for standard types such as `Integer`, `Double`, and `String` and for more complicated types such as java.io.File and java.net.URL. When we define our own data types we need to override `equals()`. Java's convention is that `equals()` must be an *equivalence relation*:

  - *Reflexive*: `x.equals(x)` is true.
  - *Symmetric*: `x.equals(y)` is true if and only if `y.equals(x)` is true.
  - *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.

  Adhering to these Java conventions can be tricky, as illustrated for Date.java and Transaction.java.

- *Memory management.* One of Java's most significant features is its ability to *automatically* manage memory. When an object can no longer be referenced, it is said to be *orphaned*. Java keeps 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*.

- *Immutability.* An *immutable* data type has the property that the value of an object never changes once constructed. By contrast, a *mutable* data type 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.

  Vector.java is an immutable data type for vectors. In order to guarantee immutability, it *defensively copies* the mutable constructor argument.

- *Exceptions and errors* are disruptive events that handle unforeseen errors *outside* our control. We have already encountered the following exceptions and errors:

  - ArithmeticException. Thrown when an exceptional arithmetic condition (such as integer division by zero) occurs.
  - ArrayIndexOutOfBoundsException. Thrown when an array is accessed with an illegal index.
  - NullPointerException. Thrown when `null` is used where an object is required.
  - OutOfMemoryError. Thrown when the Java Virtual Machine cannot allocate an object because it is out of memory.
  - StackOverflowError. Thrown when a recursive method recurs too deeply.

  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.");
```

- *Assertions* are boolean expressions which verify assumptions that we make *within* code we develop. If the expression is false, the program will terminate and report an error message. For example, suppose that you have a computed value that you might use as an index into an array. If this value were negative, it would cause a `ArrayIndexOutOfBoundsException` sometime later. But if you write the code

  ```
  assert index >= 0;
  ```

  you can pinpoint the place where the error occurred. 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.

Q + A.

Q. Are there any truly immutable classes in Java?

A. If you use reflection, you can access the `private` fields of any class and change them. Program MutableString.java demonstrates how to mutate a `String`. Program MutableInteger.java demonstrates that this is true even if the instance variable is final.

Exercises

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

4. What does the following code fragment print?

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

   Solution:

   ```
   world
   hello
   ```

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.

   Solution: `(s.length() == t.length()) && (s.concat(s).indexOf(t) >= 0)`

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);
    }
```

Solution: Reverse of the string.

13. Using our implementation of Date.java as a model, develop an implementation of Transaction.java.

14. Using our implementation of `equals()` in Date.java as a model, develop an implementation of `equals()` for Transaction.java.

Creative Problems

16. **Rational numbers.** Implement an immutable data type Rational.java 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, but use as instance variables two `long` values that represent the numerator and denominator to limit the possibility of overflow. Use Euclid's algorithm to ensure that the numerator and denominator never have any common factors. Include a test client that exercises all your methods.

18. **Sample variance for accumulator.** Validate that the following code, which adds the methods `var()` and `stddev()` to Accumulator.java, computes the mean, sample variance, and sample standard deviation of the numbers presented as arguments to `addDataValue()`.

Reference: Here is a good explanation of this one-pass method, that was first discovered by Welford in 1962. This approach can be applied to computing the skewness, kurtosis, regression coefficients, and Pearson's correlation coefficient.

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

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

Last modified on August 26, 2016.