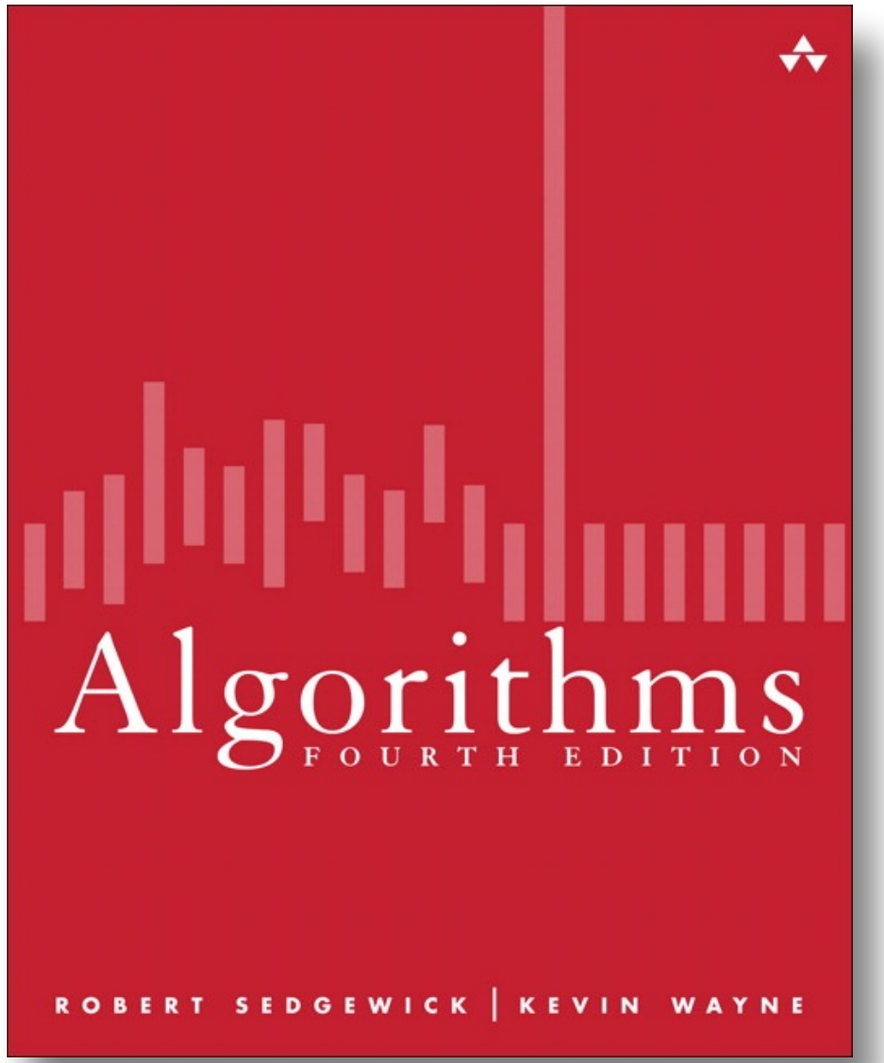
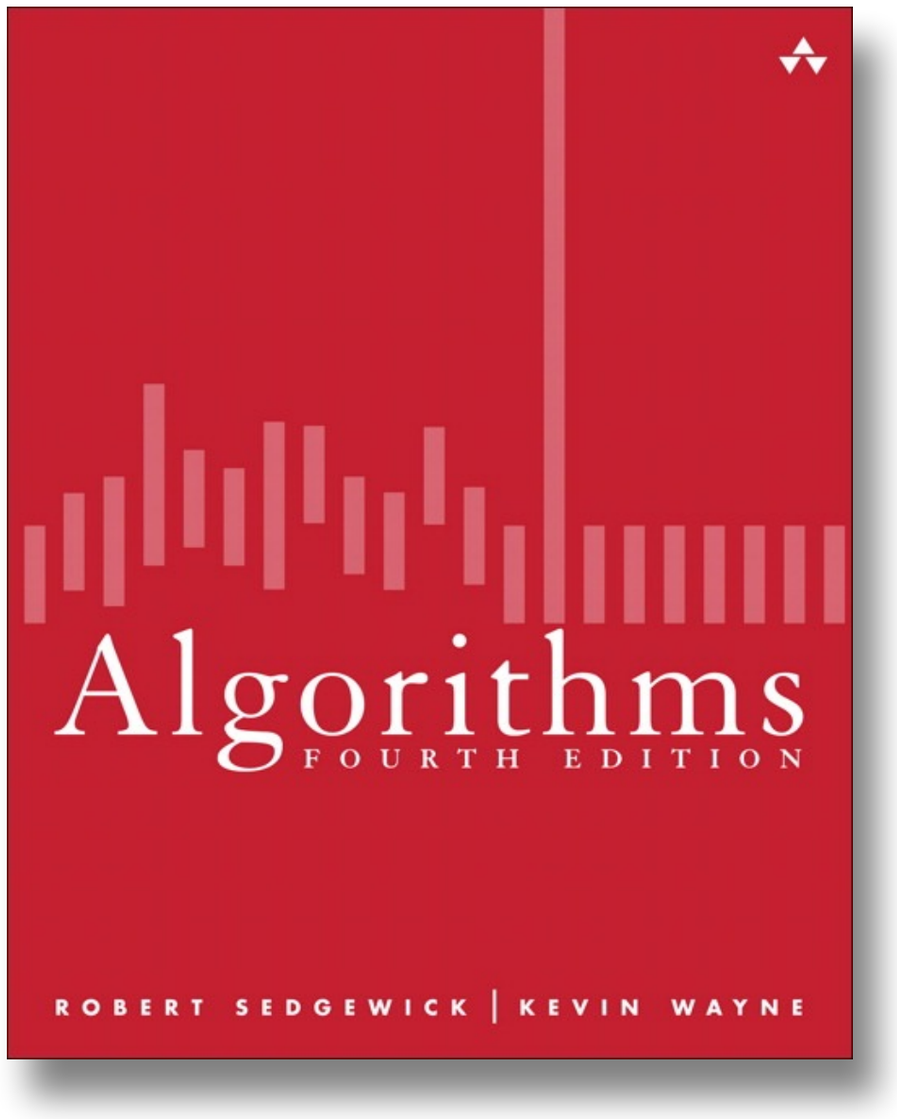
CSC 402

Data Structures I
Lecture 1



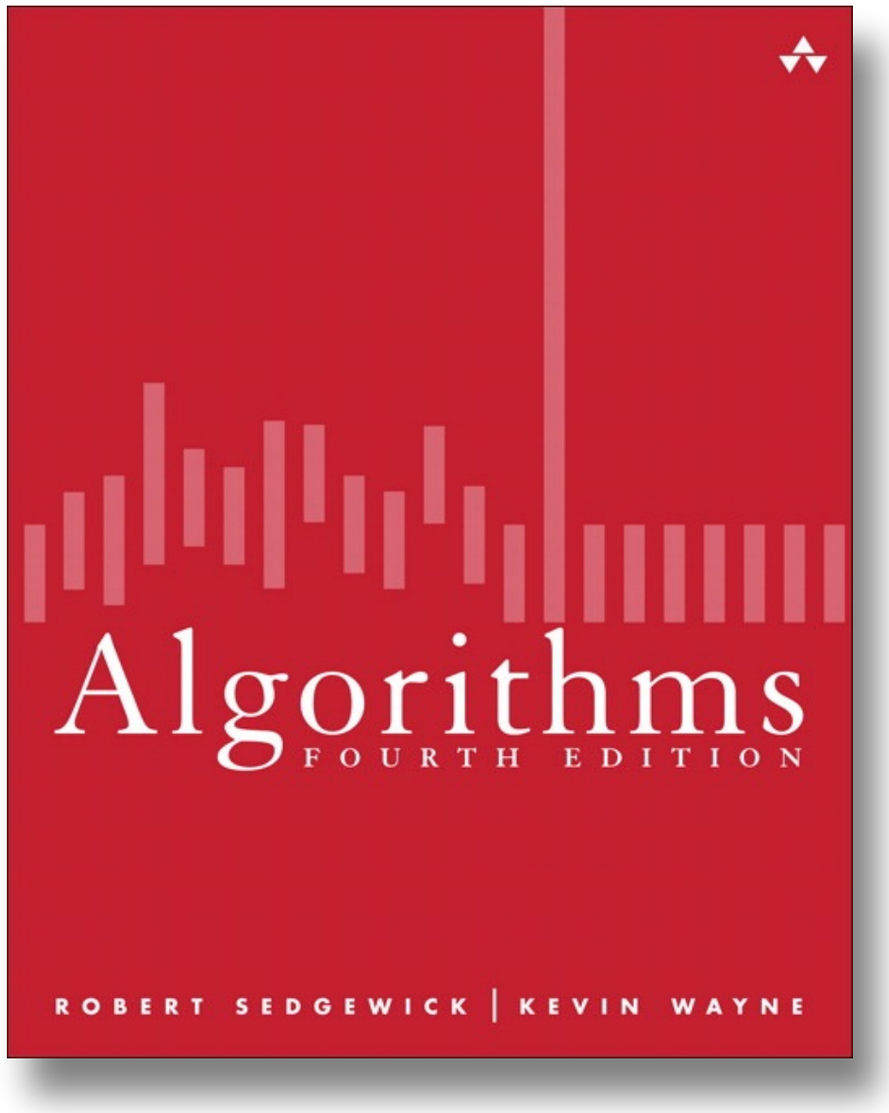
Algorithms, 4/E
Robert Sedgewick, Princeton University
Kevin Wayne
ISBN-10: 032157351X • ISBN-13: 9780321573513
©2011 • Addison-Wesley Professional, 992 pp
Published 03/09/2011

Contact Info



ebacklin@depaul.edu
(708)235-1973

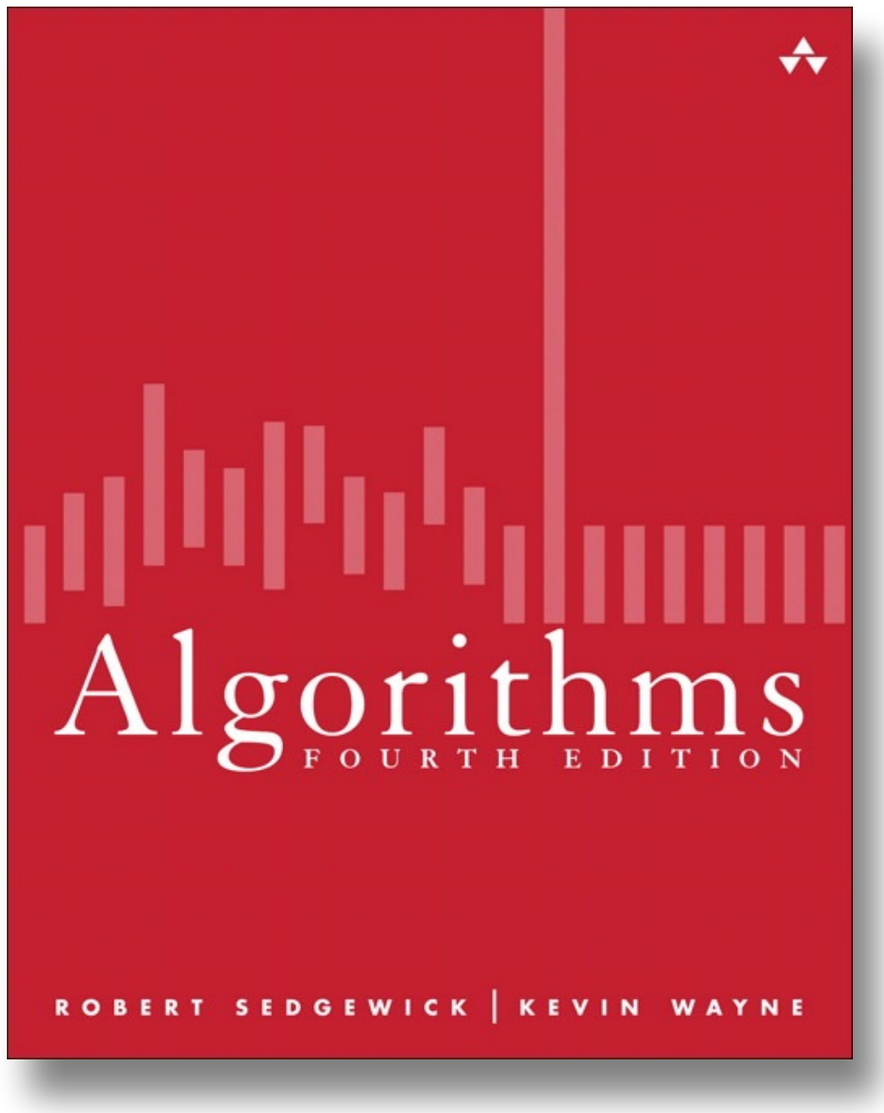
Assignments and Lectures



Lecture notes for each lecture, and the files studied in that lecture, will be posted before the lecture in the D2L section for the associated week.

Usually the notes will be posted the day before class or earlier.

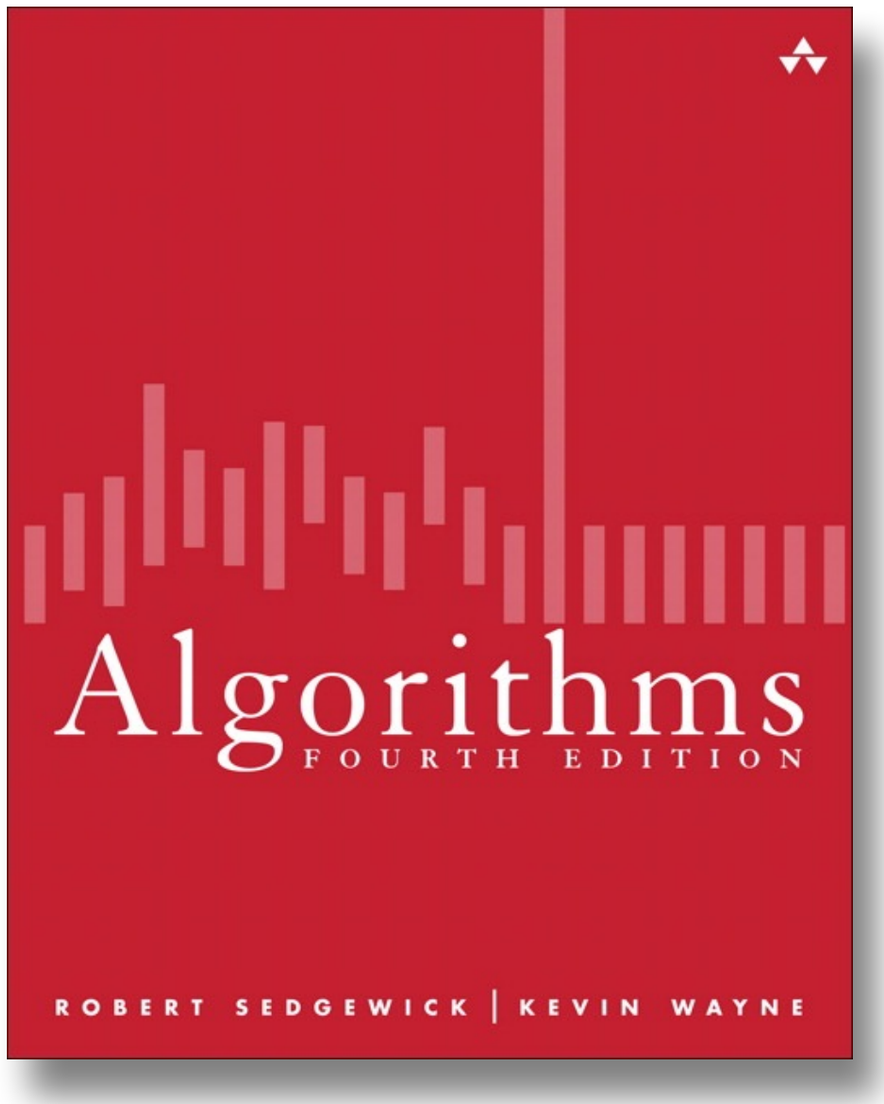
Submitting Assignments



Please follow the following procedures to ensure that you get proper credit for all assignments.

- On your computer create a folder (Your Last Name)_Week1 (for week 1 it will change from week to week)
- Within this folder (Your Last Name)_Week1, create a folder for the assignment. For example there were 3 assignments this week, so you create 3 folders, Assigment1, Assignment2 and Assignment3. Your directory will be Backlin_Week1 at the top, and Assignment1, Assignment2, Assignment3 under it as sub-folders.
- Place your java programs and the screenshot of the program results when you run it in each of the associated sub folders.

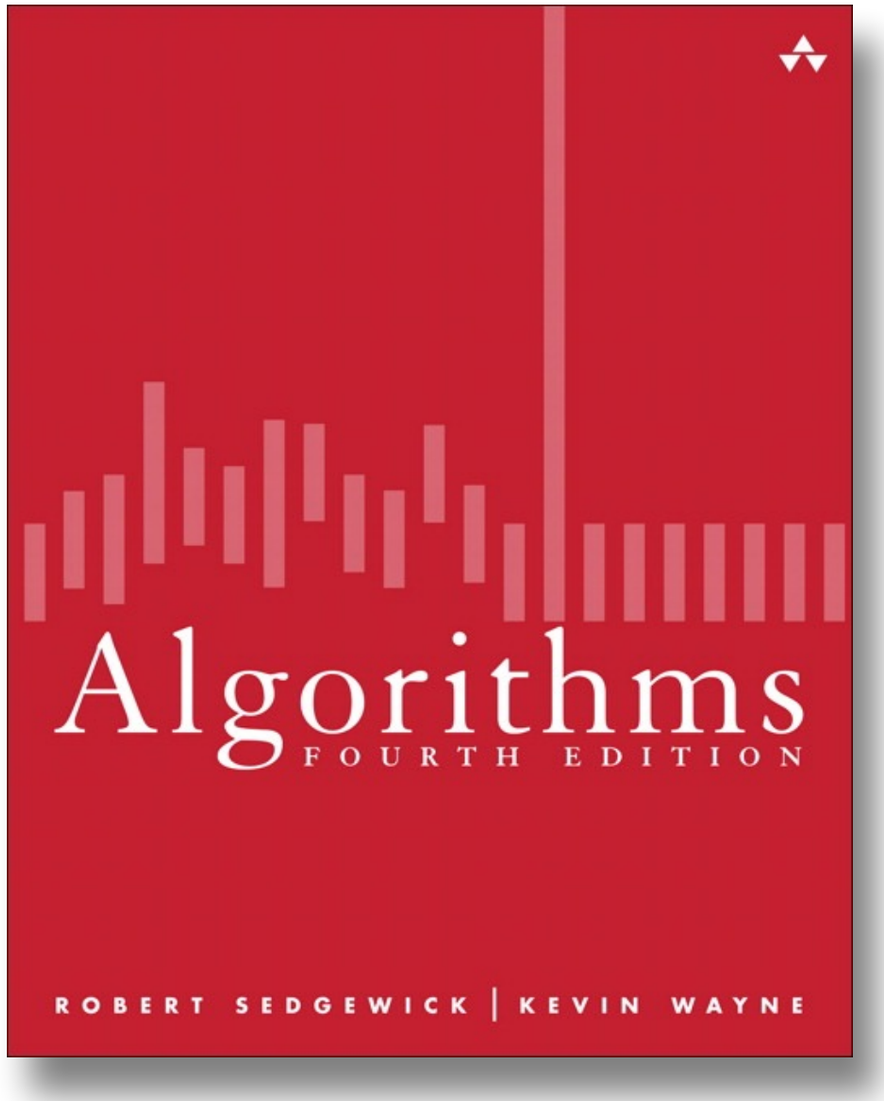
Submitting Assignments



So for each assignment you will:

- run the program and take a screenshot of the results, please make them jpg, bmp or paste them into a word doc.
- put both the Java program and screenshot you just took into the correct Assignment folder.
- Now after you have completed all the assignments and included all files within each Assignment folders, zip up the top folder, that being (Your Last Name)_Week1 (for Week 1)

Assignments

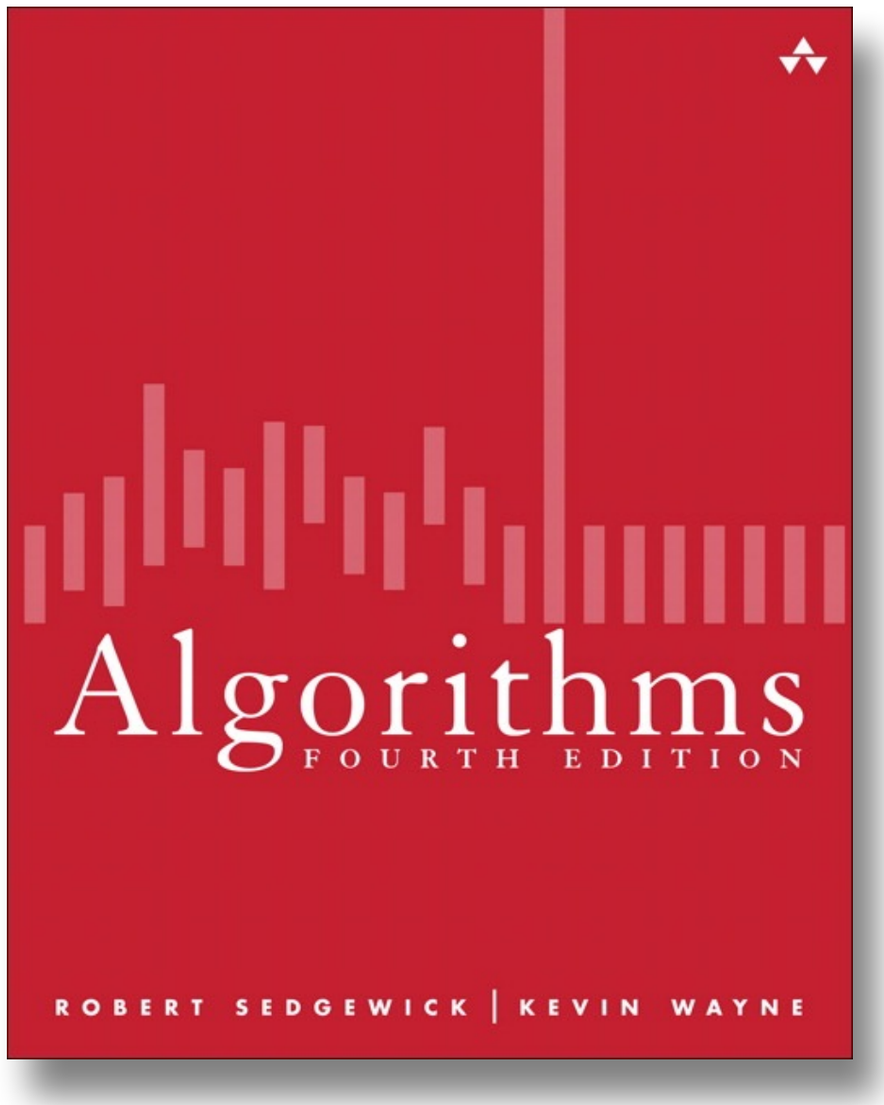


Assignments will be posted almost every week, starting this week, and will be due 11:59 the following Monday.

The assignments provide the practice needed to understand and master the material. It will be hard to do well on the exams without doing the assignments and it will be very hard to get a good grade if you do poorly on the assignments.

Plagiarized solutions from the lecture slides, that being using my solutions, will not be accepted.

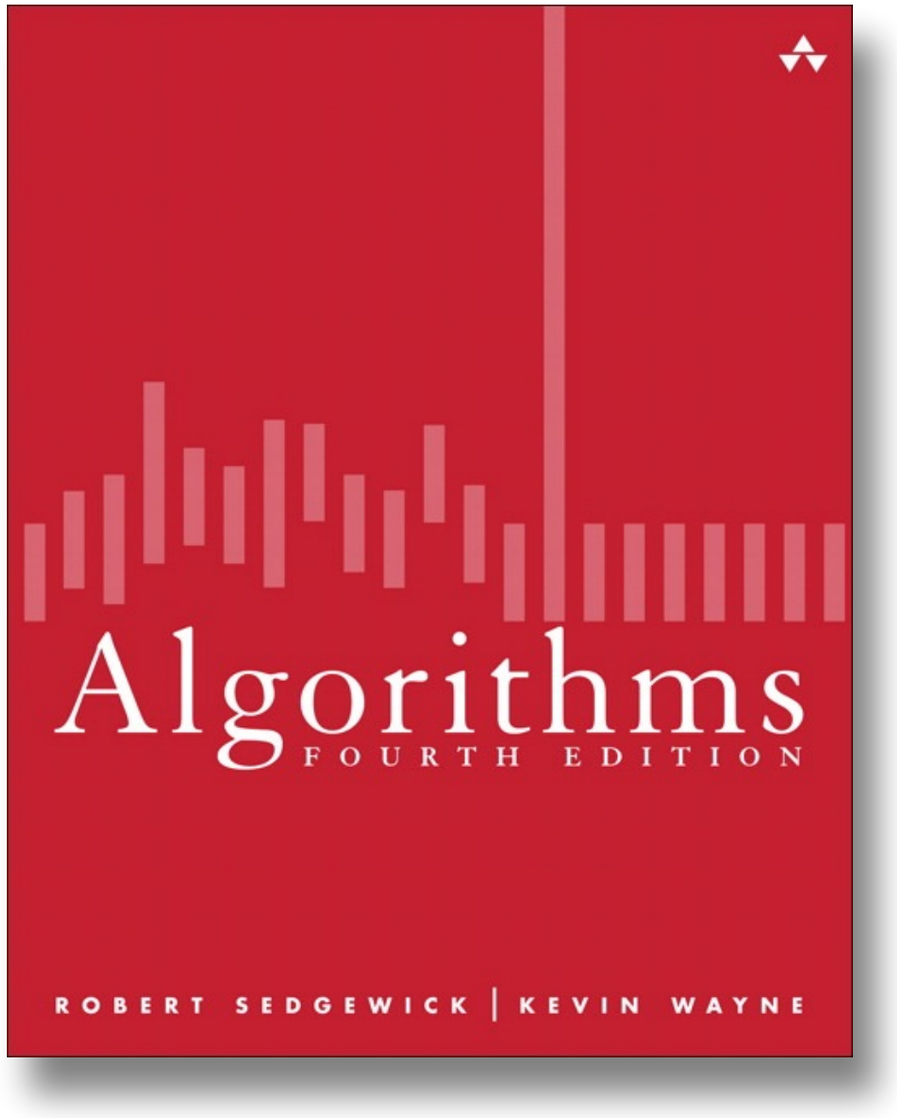
Assignments



Plagiarism includes, but is not limited to, the following: copying part or all of another person's program, starting with another person's program and modifying it to submit as your own, allowing or hiring another person to write part or all of a program for you, and so on.

If you have any questions or doubts about what plagiarism is, you should consult me. You should be familiar with and to adhere to DePaul's Academic Integrity Policy.

Assignments



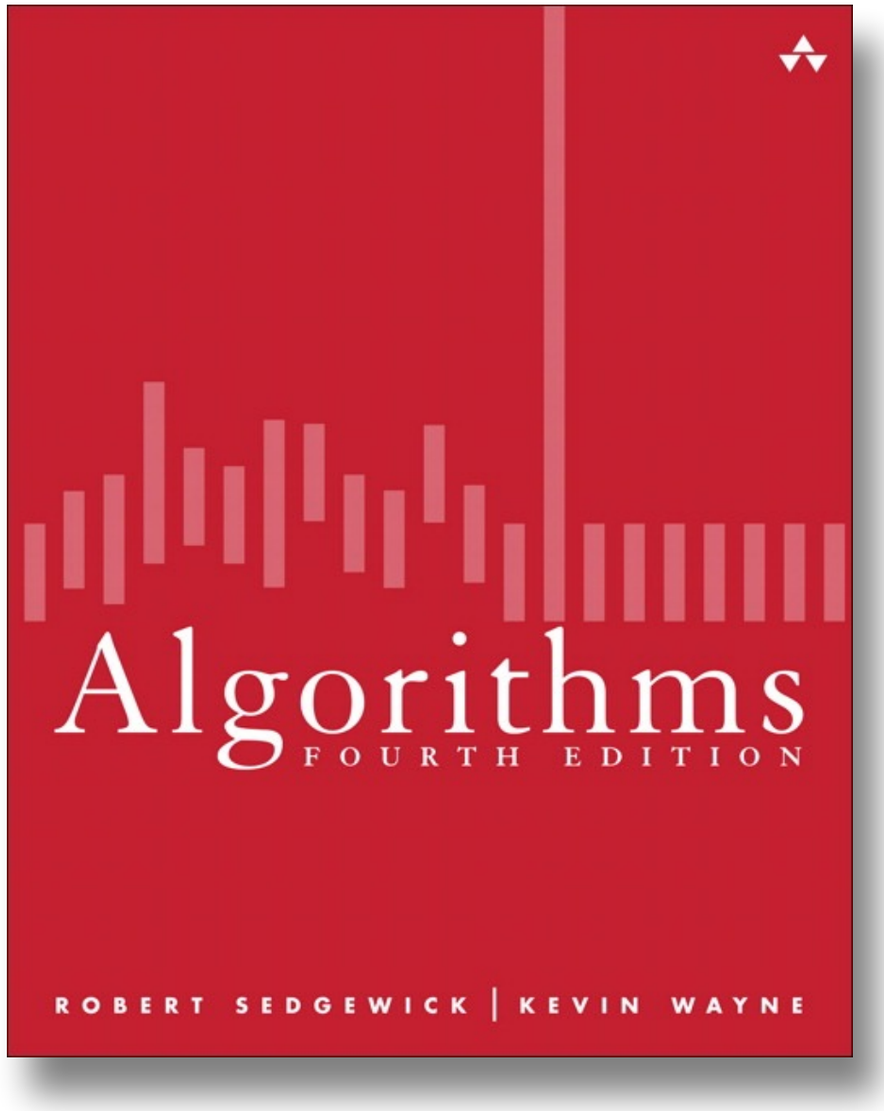
Assignment totals 180 points
Midterm totals 60 points
Final totals 90 points

Grading

Breakdown

Assignments:	50%
Midterm Exam:	20%
Final Exam:	30%

Assignments

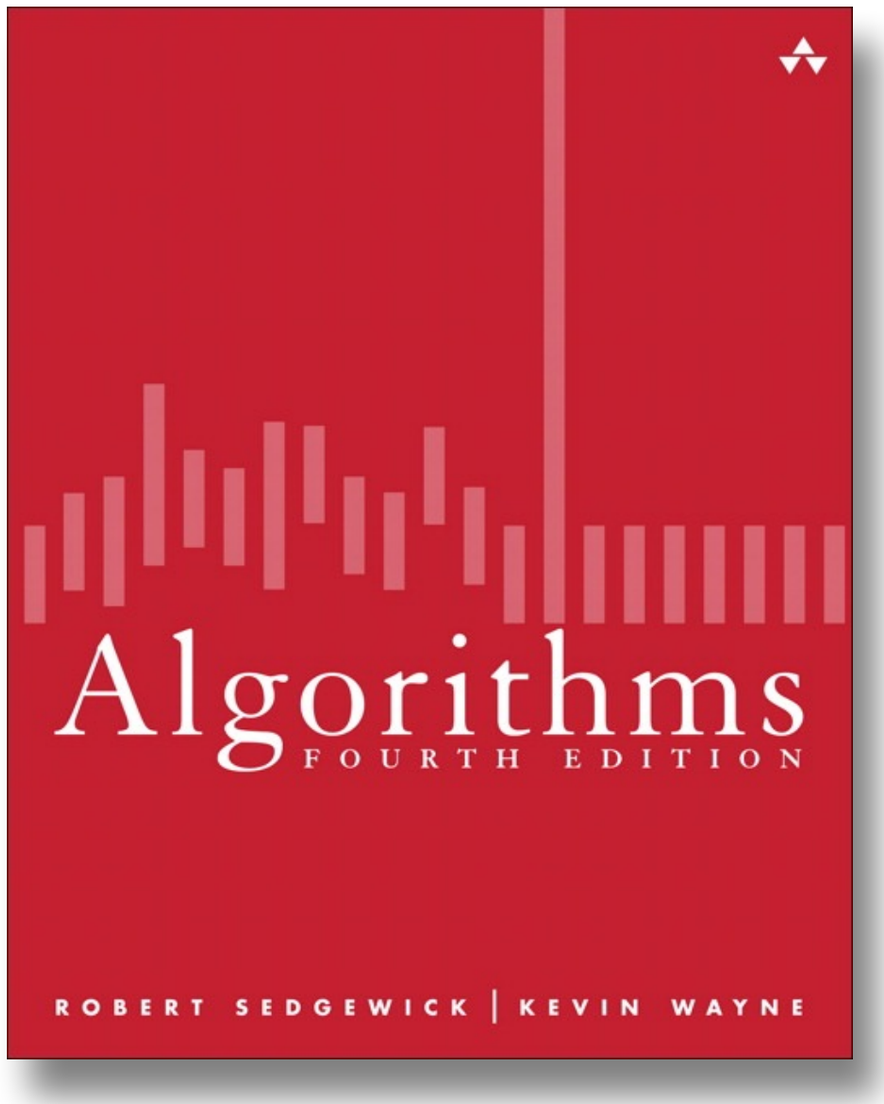


Assignment totals 180 points
Midterm totals 60 points
Final totals 90 points

Grading Scale

93-100	A
90-92.99	A-
87-89.99	B+
83-86.99	B
80-82.99	B-
77-89.99	C+
73-76.99	C
70-72.99	C-
67-69.99	D+
60-66.99	D
0-59.99	F

1. Readings - Readings are assigned from the textbook and web sources given.
2. Assignments - Assignments are Java programs
3. Exam - Midterm and Final



Language Comparisons

- Python - Java

Python/Java - (Main)

Python

```
01 print ("Hello")
```

Java

```
01 package algs11;
02 public class Hello {
03   public static void main (String[] args) {
04     System.out.println ("Hello");
05   }
06 }
```

- Every java program requires a main function. It must be declared exactly as on the third line above.
- Every java function resides in a class, in this case Hello.
- Every java class resides in a package, in this case algs11.
- Java includes visibility annotations, such as public and private. In python, everything is public.
- Java functions are written using the keyword static. (More later.)
- Java functions must include types for arguments and results, in this case String[] and void.

Python/Java - (Main)

Python

```
01 print ("Hello")
```

Java

```
01 package algs11;
02 public class Hello {
03   public static void main (String[] args) {
04     System.out.println ("Hello");
05   }
06 }
```

Both java and python code is compiled before it is run.

- The python interpreter performs both these steps, interactively.
- For java, we will be using eclipse, which continuously compiles your code. If the code successfully compiles (no red marks), then you can run it by pressing the play button.

Python/Java - (Import)

Python

```
01 print ("Hello")
```

Java

```
01 package algs11;
02 import stdlib.*;
03 public class Hello {
04   public static void main (String[] args) {
05     StdOut.println ("Hello");
06   }
07 }
```

System.out and System.in refer to objects.

StdOut is a class in the stdlib package.
System is a class in the java.lang package.

All of the classes in java.lang are imported implicitly into every java program. All other classes must be imported explicitly.

Python/Java - (Import)

Python

```
01 print ("Hello")
```

Java

```
01 package algs11;
02 import stdlib.*;
03 public class Hello {
04   public static void main (String[] args) {
05     StdOut.println ("Hello");
06   }
07 }
```

The statement import stdlib.* makes the classes declared in stdlib visible in our program.

- Remove the import statement and see what happens.

Instead of importing every class from a package, you can also import a single class.

- Try replacing import stdlib.* with import stdlib.StdOut.

Unnecessary imports will generate a warning.

- Try adding import stdlib.StdIn.

If your code has a compiler error, it will not run.
If your code has a warning, you can run it. But you should fix the warning.

Python/Java - (Fully Qualified Class Names)

Python

```
01 print ("Hello")
```

Java

```
01 package algs11;
02 public class Hello {
03   public static void main (java.lang.String[] args) {
04     stdlib.StdOut.println ("Hello");
05   }
06 }
```

As an alternative to using import, you can also use a class's fully qualified name, which includes the package explicitly.

Fully qualified names make code rather verbose, so usually people prefer to use import.

Python/Java - (Variable Declarations)

Python

```
01 name = "Bob"
02 print ("Hello " + name)
```

Java

```
01 package algs11;
02 import stdlib.*;
03 public class Hello {
04   public static void main (String[] args) {
05     String name;
06     name = "Bob";
07     StdOut.println ("Hello " + name);
08   }
09 }
```

In python, values are typed, but variables are not.
In java, both values and variables are typed. Variable types must be explicitly declared.

The declaration and initialization can be combined into a single statement.

```
01  String name = "Bob";
02  StdOut.println ("Hello " + name);
```

In both languages, + is used to represent string concatenation.

Python/Java - (Variable Declarations)

Python

```
01 name = "Bob"
02 print ("Hello ", end="")
03 print (name)
```

Java

```
01 package algs11;
02 import stdlib.*;
03 public class Hello {
04   public static void main (String[] args) {
05     String name = "Bob";
06     StdOut.print ("Hello ");
07     StdOut.println (name);
08   }
09 }
```

Here's another version which does not use concatenation.

Python/Java - (Types)

Python

```
01 x = "Bob"
02 print ("Hello " + x)
03 x = 42
04 print (x - 1)
```

Java

```
01 package algs11;
02 import stdlib.*;
03 public class Hello {
04   public static void main (String[] args) {
05     String x;
06     x = "Bob";
07     StdOut.println ("Hello " + x);
08     x = 42; // Compiler error
09     StdOut.println (x - 1); // Compiler error
10   }
11 }
```

Python allows a single variable to be used at multiple types.

By typing variables, java catches more errors before runtime.

Python/Java - (Syntax)

Python

```
01 name = "Bob"
02 print ("Hello ", end="")
03 print (name)
```

Java

```
01 package algs11;
02 import stdlib.*;
03 public class Hello {
04   public static void main (String[] args) {
05     String name = "Bob";
06     StdOut.print ("Hello ");
07     StdOut.println (name);
08   }
09 }
```

Java uses semicolons and curly-braces, where python uses newlines, colons and indentation.
When formatting java, the conventions for indentation and newlines mimic those of python. But in java, these are just conventions, not requirements.

Try removing the newlines in the python and java code above.

Python/Java - (Function Declarations)

Python

```
01 def addHello (x):
02   return "Hello " + x
03
04 def main ():
05   print (addHello ("Bob"))
06   print (addHello ("Alice"))
07
08 main ()
```

Java

```
01 package algs11;
02 import stdlib.*;
03 public class Hello {
04   public static String addHello (String x) {
05     return "Hello " + x;
06   }
07   public static void main (String[] args) {
08     addHello ("Bob");
09     addHello ("Alice");
10   }
11 }
```

Java requires declaration of return type and parameter types.

Python/Java - (Scope)

Python

```
01 def declareX ():
02   global x
03   x = 0
04
05 def useX ():
06   x
07
08 useX () # runtime error
09 declareX ()
10 useX () # no problems
```

Java

```
01 package algs11;
02 import stdlib.*;
03 public class Hello {
04   public static void declareX () {
05     int x;
06   }
07   public static void useX () {
08     x;  // compiler error
09   }
10   public static void main (String[] args) {
11     useX ();
12     declareX ();
13     useX ();
14   }
15 }
```

Java compiler removes names for variables, replacing them with numbers (offsets in memory). This is one characteristic of “static” languages.

Python keeps names for variables at runtime. As a result, python uses hash tables to store variable/value pairs. This is characteristic of “dynamic” languages.
09}

Java's approach is more efficient. Python's is more flexible.

“Scripting languages”, such as perl and javascript, use the python approach. Most other languages, including C, C++, C#, Objective-C, Swift and FORTRAN, use the java approach.

Python/Java - (Primitive vs Object)

Python

```
01 i = 0
02 while (i < 3):
03   print ("Hello")
04   i = i + 1
```

Java

```
01 package algs11;
02 import stdlib.*;
03 public class Hello {
04   public static void main (String[] args) {
05     int i = 0;
06     while (i < 3) {
07       StdOut.print ("Hello");
08       i = i + 1;
09     }
10   }
11 }
```

In python, everything is an object. This makes the language quite simple, but comes at a cost.

If you change int to Integer in the program above, then it will run like a python program.

Integer is an object type, whereas int is a primitive type.

Primitive values can be directly manipulated by the arithmetic hardware of the computer, whereas object values cannot.

Python/Java - (Primitive vs Object)

```
01 package algs11;
02 import stdlib.*;
03 public class Hello {
04   public static void main (String[] args) {
05     Integer i = Integer.valueOf (0);
06     while (i.intValue () < 3) {
07       StdOut.println ("Hello");
08       int tmp = i.intValue () + 1;
09       i = Integer.valueOf (tmp);
10     }
11   }
12 }}
```

If we use Integer, there will be many implicit conversions. Here is an equivalent program with the conversions made explicit.

Python/Java - (Primitive vs Object)

- In addition to having lots of extra function and method calls, this code potentially creates a bunch of objects. There are four separate objects, holding the values 0 through 3.
- Conversion from base type to object type is called boxing. In the code above, this is achieved by the calls to Integer.valueOf.
- Conversion from object type to base type is called unboxing. In the code above, this is achieved by the calls to i.intValue.
- We will mostly only see code boxing integers and doubles. Here is a table summarizing the operations.

Base type	Object type	Boxing (base to object)	Unboxing (object to base)
`int base = 0;`	`Integer object = null;`	`object = Integer.valueOf(base);`	`base = object.intValue();`
`double base = 0.0;`	`Double object = null;`	`object = Double.valueOf(base);`	`base = object.doubleValue();`

Java has five additional base types, as follows.

Base type	Object type	Boxing (base to object)	Unboxing (object to base)
`boolean base = false;`	`Boolean object = null;`	`object = Boolean.valueOf(base);`	`base = object.booleanValue();`
`float base = 0.0F;`	`Float object = null;`	`object = Float.valueOf(base);`	`base = object.floatValue();`
`byte base = 0;`	`Byte object = null;`	`object = Byte.valueOf(base);`	`base = object.byteValue();`
`char base = 0;`	`Character object = null;`	`object = Character.valueOf(base);`	`base = object.charValue();`
`short base = 0;`	`Short object = null;`	`object = Short.valueOf(base);`	`base = object.shortValue();`
`long base = 0L;`	`Long object = null;`	`object = Long.valueOf(base);`	`base = object.longValue();`

Python/Java - (Objects and Equality)

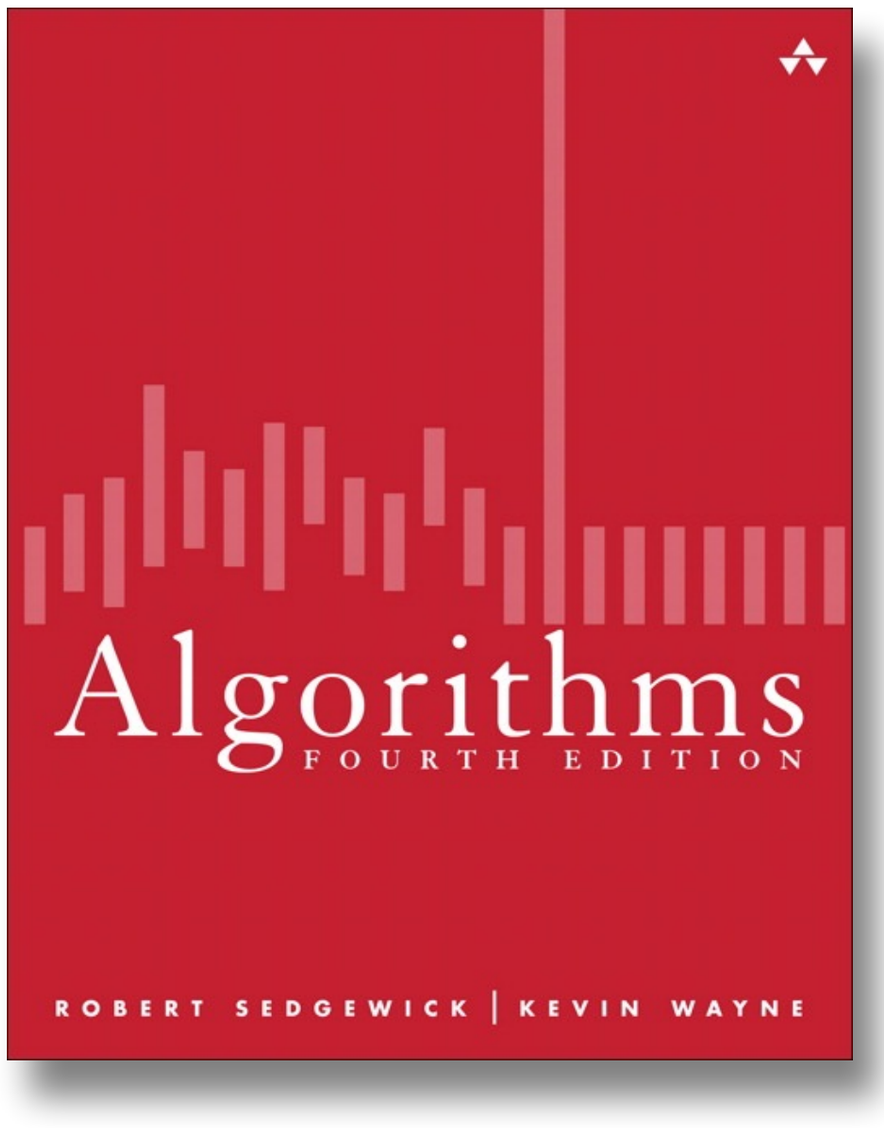
```
01 package algs11;
02 import stdlib.*;
03 public class Hello {
04   public static void main (String[] args) {
05     Integer i = Integer.valueOf (0);
06     while (i.intValue () < 3) {
07       StdOut.println ("Hello");
08       int tmp = i.intValue () + 1;
09       i = Integer.valueOf (tmp);
10     }
11   }
12 }}
```

In Java, the == operator checks object identity on object types. That is, the two operands refer to the same object. More concretely: the two operands evaluate to the same address in memory.

Unlike other languages (such as C++) this behavior cannot be changed.

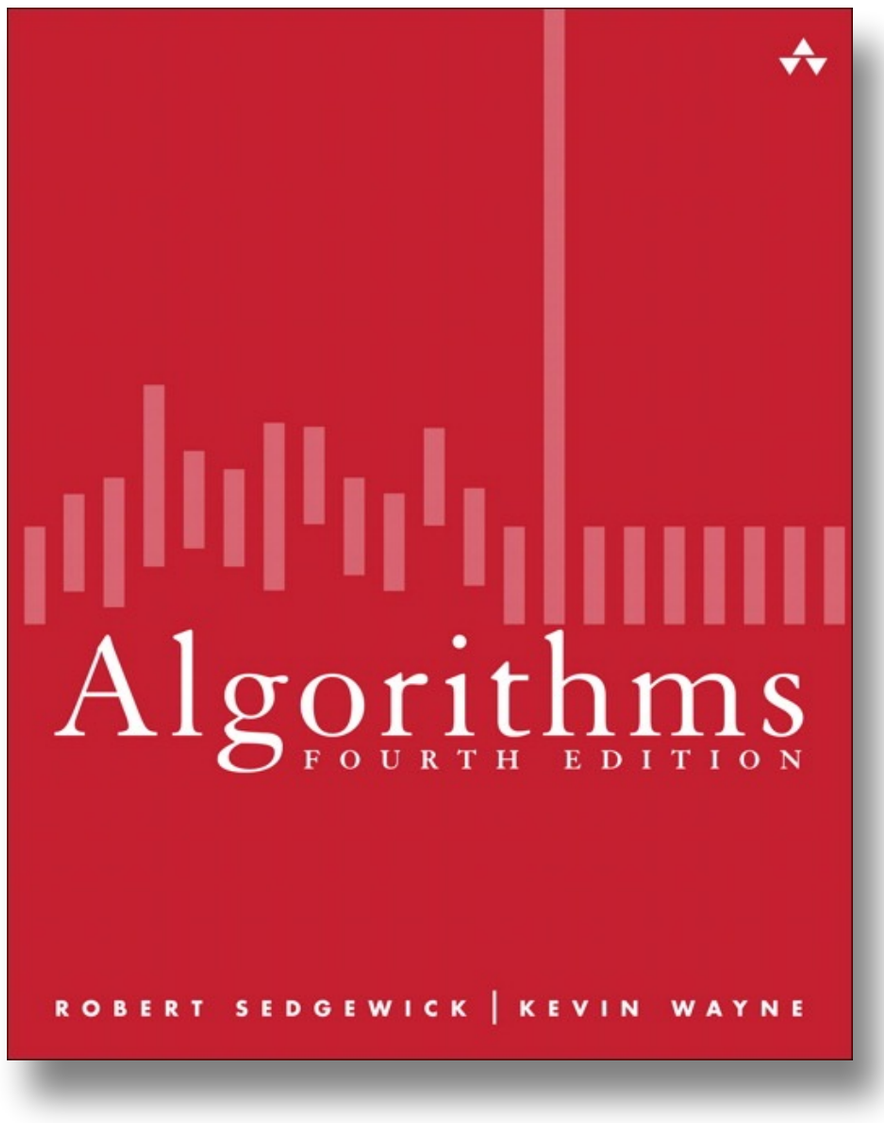
Objects all have an equals method. The behavior of equals varies from class to class. The default method, defined in java.lang.Object, tests identity, just like ==. Many of java's builtin classes override this default behavior to check value equality rather than object identity.

Lecture Overview



1.1 Programming Model

- Primitive data types and expressions
- Statements
- Static Methods



1.1 Programming Model

- Primitive data types and expressions
- Statements
- Static Methods

Primitive Data Types and Expressions

Our study of algorithms is based upon implementing them as programs written in the Java programming language. We do so for several reasons:

- Our programs are concise, elegant, and complete descriptions of algorithms.
- You can run the programs to study properties of the algorithms.
- You can put the algorithms immediately to good use in applications.

A data type is a set of values and a set of operations on those values. The following four primitive data types are the basis of the Java language:

- Integers, with arithmetic operations (int)
- Real numbers, again with arithmetic operations (double)
- Booleans, the set of values { true, false } with logical operations (boolean)
- Characters, the alphanumeric characters and symbols that you type (char)

Primitive Data Types and Expressions

A Java program manipulates variables that are named with identifiers. Each variable is associated with a data type and stores one of the permissible data-type values. We use expressions to apply the operations associated with each type.

term	examples	definition
primitive data type	`int double boolean char`	a set of values and a set of operations on those values (built in to the Java language)
identifier	`a abc Ab$ a_b ab123 lo hi`	a sequence of letters, digits, _, and $, the first of which is not a digit
variable	[*any identifier*]	names a data-type value
operator	`+ - * /`	names a data-type operation

literal	`int` `1 0 -42`
	`double` `2.0 1.0e-15 3.14`
	`boolean` `true false`
	`char` `'a' '+' '9' '\n'`
		source-code representation of a value

expression	`int` `lo + (hi - lo)/2`
	`double` `1.0e-15 * t`
	`boolean` `lo <= hi`
		a literal, a variable, or a sequence of operations on literals and/or variables that produces a value

Primitive Data Types and Expressions

The following table summarizes the set of values and most common operations on those values for Java's int, double, boolean, and char data types.

type	set of values	operators	typical expressions	
			expression	value
`int`	integers between -2^{31} and $+2^{31}-1$ (32-bit two's complement)	+ (add)	`5 + 3`	8
		- (subtract)	`5 - 3`	2
		* (multiply)	`5 * 3`	15
		/ (divide)	`5 / 3`	1
		% (remainder)	`5 % 3`	2
`double`	double-precision real numbers (64-bit IEEE 754 standard)	+ (add)	`3.141 - .03`	3.111
		- (subtract)	`2.0 - 2.0e-7`	1.9999998
		* (multiply)	`100 * .015`	1.5
		/ (divide)	`6.02e23 / 2.0`	3.01e23
`boolean`	`true` or `false`	&& (and)	`true && false`	`false`
		|| (or)	`false || true`	`true`
		! (not)	`!false`	`true`
		^ (xor)	`true ^ true`	`false`
`char`	characters (16-bit)	[*arithmetic operations, rarely used*]		

Primitive Data Types and Expressions

Expressions - Typical expressions are infix. When an expression contains more than one operator, the precedence order specifies the order in which they are applied: The operators * and / (and %) have higher precedence than (are applied before) the + and - operators; among logical operators, ! is the highest precedence, followed by && and and then ||. Generally, operators of the same precedence are left associative (applied left to right). You can use parentheses to override these rules.

Type conversion - Numbers are automatically promoted to a more inclusive type if no information is lost.For example, in the expression 1 + 2.5, the 1 is promoted to the double value 1.0 and the expression evaluates to the double value 3.5. A cast is a directive to convert a value of one type into a value of another type. For example (int) 3.7 is 3. Casting a double to an int truncates toward zero.

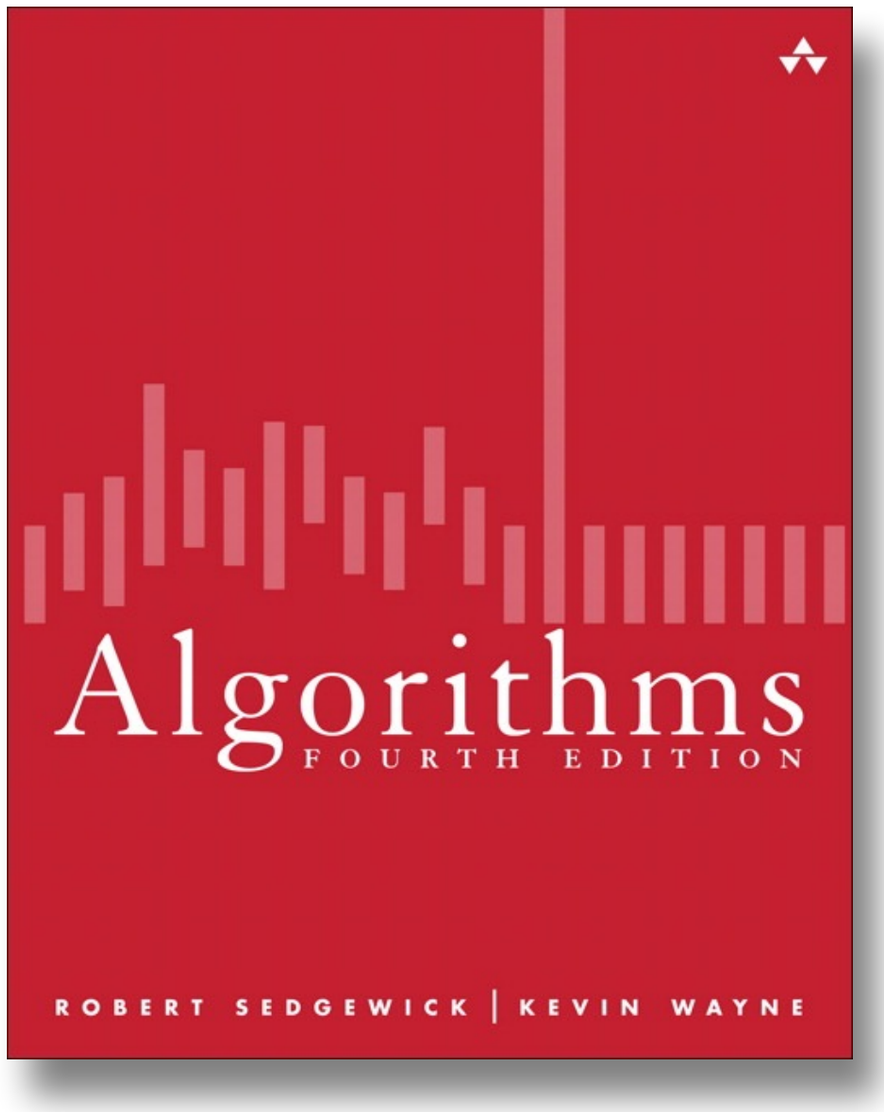
Primitive Data Types and Expressions

Comparisons - The following mixed-type operators compare two values of the same type and produce a boolean value:

- equal (==)
- not equal (!=)
- less than (<)
- less than or equal (<=)
- greater than (>)
- greater than or equal (>=)

Other primitive types - Java's int has a 32-bit representation; Java's double type has a 64-bit representation. Java has five additional primitive data types:

- 64-bit integers, with arithmetic operations (long)
- 16-bit integers, with arithmetic operations (short)
- 16-bit characters, with arithmetic operations (char)
- 8-bit integers, with arithmetic operations (byte)
- 32-bit single-precision real numbers, with arithmetic operations (float)



1.1 Programming Model

- Primitive data types and expressions
- Statements
- Static Methods

Statements

Statements - A Java program is composed of statements, which define the computation by creating and manipulating variables, assigning data-type values to them, and controlling the flow of execution of such operations.

- Declarations create variables of a specified type and name them with identifiers. Java is a strongly typed language because the Java compiler checks for consistency. The scope of a variable is the part of the program where it is defined.

- Assignments associate a data-type value (defined by an expression) with a variable.

- Initializing declarations combine a declaration with an assignment to initialize a variable at the same time it is declared.

- Implicit assignments. The following shortcuts are available when our purpose is to modify a variable's value relative to the current value:
  - Increment/decrement operators: the code i++ is shorthand for i = i + 1. The code ++i is the same except that the expression value is taken after the increment/decrement, not before.
  - Other compound operators: the code i /= 2 is shorthand for i = i/2.

Statements

Statements - A Java program is composed of statements, which define the computation by creating and manipulating variables, assigning data-type values to them, and controlling the flow of execution of such operations.

- Initializing declarations combine a declaration with an assignment to initialize a variable at the same time it is declared.

- Conditionals provide for a simple change in the flow of execution—execute the statements in one of two blocks, depending on a specified condition.

- Loops provide for a more profound change in the flow of execution—execute the statements in a block as long as a given condition is true. We refer to the statements in the block in a loop as the body of the loop.

- Break and continue. Java supports two additional statements for use within while loops:
  - The break statement, which immediately exits the loop
  - The continue statement, which immediately begins the next iteration of the loop

- For notation . Many loops follow this scheme: initialize an index variable to some value and then use a while loop to test a loop continuation condition involving the index variable, where the last statement in the while loop increments the index variable. You can express such loops compactly with Java's for notation.

- Single-statement blocks. If a block of statements in a conditional or a loop has only a single statement, the curly braces may be omitted.

Statements

The following table illustrates different kinds of Java statements.

statement	examples	definition
declaration	`int i;` `double c;`	create a variable of a specified type, named with a given identifier
assignment	`a = b + 3;` `discriminant = b*b - 4.0*c;`	assign a data-type value to a variable
initializing declaration	`int i = 1;` `double c = 3.141592625;`	declaration that also assigns an initial value
implicit assignment	`i++;` `i += 1;`	`i = i + 1;`
conditional (`if`)	`if (x < 0) x = -x;`	execute a statement, depending on boolean expression
conditional (`if-else`)	`if (x > y) max = x;` `else max = y;`	execute one or the other statement, depending on boolean expression

loop (`while`)	`int v = 0;`
	`while (v <= N)`
	`  v = 2*v;`
	`double t = c;`
	`while (Math.abs(t - c/t) > 1e-15*t)`
	`  t = (c/t + t) / 2.0;`
		execute statement until boolean expression is `false`

loop (`for`)	`for (int i = 1; i <= N; i++)`
	`  sum += 1.0/i;`
	`for (int i = 0; i <= N; i++)`
	`  StdOut.println(2*Math.PI*i/N);`
		compact version of `while` statement

call	`int key = StdIn.readInt();`	invoke other methods (see page 22)
return	`return false;`	return from a method (see page 24)

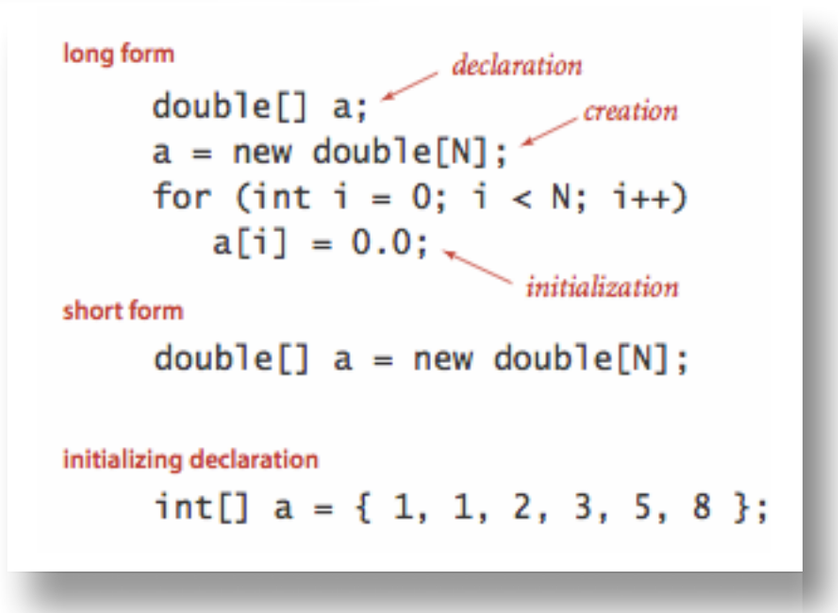
Arrays

Arrays - An array stores a sequence of values that are all of the same type. If we have N values, we can use the notation a[i] to refer to the ith value for any value of i from 0 to N-1.

- Creating and initializing an array. Making an array in a Java program involves three distinct steps:
  - Declare the array name and type.
  - Create the array.
  - Initialize the array values.

- Default array initialization. For economy in code, we often take advantage of Java's default array initialization convention and combine all three steps into a single statement. The default initial value is zero for numeric types and false for type boolean.

- Initializing declaration. We can specify the initialization values at compile time, by listing literal values between curly braces, separated by commas.



Arrays

Arrays - An array stores a sequence of values that are all of the same type. If we have N values, we can use the notation a[i] to refer to the ith value for any value of i from 0 to N-1.

- Using an array. Once we create an array, its size is fixed. A program can refer to the length of an array a[] with the code a.length. Java does automatic bounds checking—if you access an array with an illegal index your program will terminate with an ArrayIndexOutOfBoundsException.

- Aliasing. An array name refers to the whole array—if we assign one array name to another, then both refer to the same array, as illustrated in the following code fragment.

```
int[] a = new int[N];
...
a[i] = 1234;
...
int[] b = a;
...
b[i] = 5678;   // a[i] is now 5678.
```

- This situation is known as aliasing and can lead to subtle bugs.

- Two-dimensional arrays. A two-dimensional array in Java is an array of one-dimensional arrays. A two-dimensional array may be ragged (its arrays may all be of differing lengths), but we most often work with (for appropriate parameters M and N) M-by-N two-dimensional arrays. To refer to the entry in row i and column j of a two-dimensional array a[][], we use the notation a[i][j].

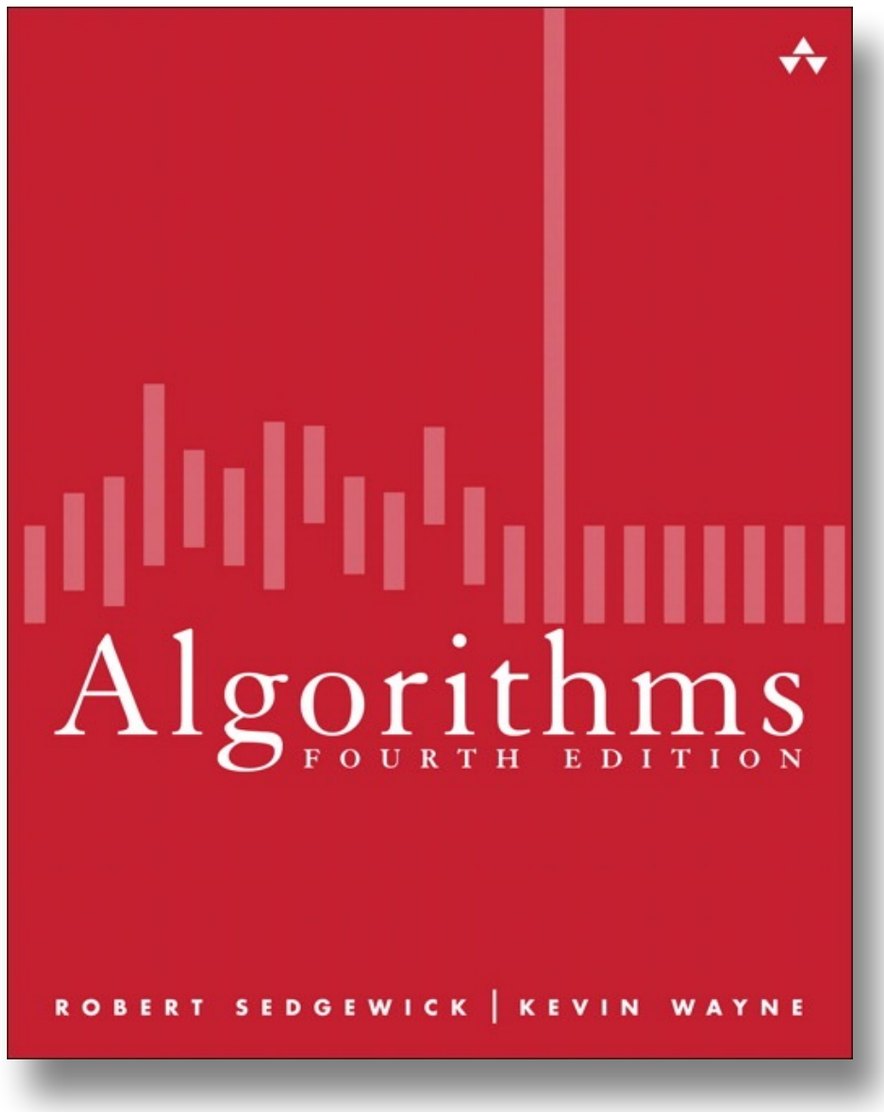
Symbol Tables

- **Insert** a value with specified key.
- Given a key, **search** for the corresponding value.

Ex. DNS lookup.

- Insert domain name with specified IP address.
- Given domain name, find corresponding IP address.

domain name	IP address
www.cs.princeton.edu	128.112.136.11
www.princeton.edu	128.112.128.15
www.yale.edu	130.132.143.21
www.harvard.edu	128.103.060.55
www.simpsons.com	209.052.165.60
↑ key	↑ value



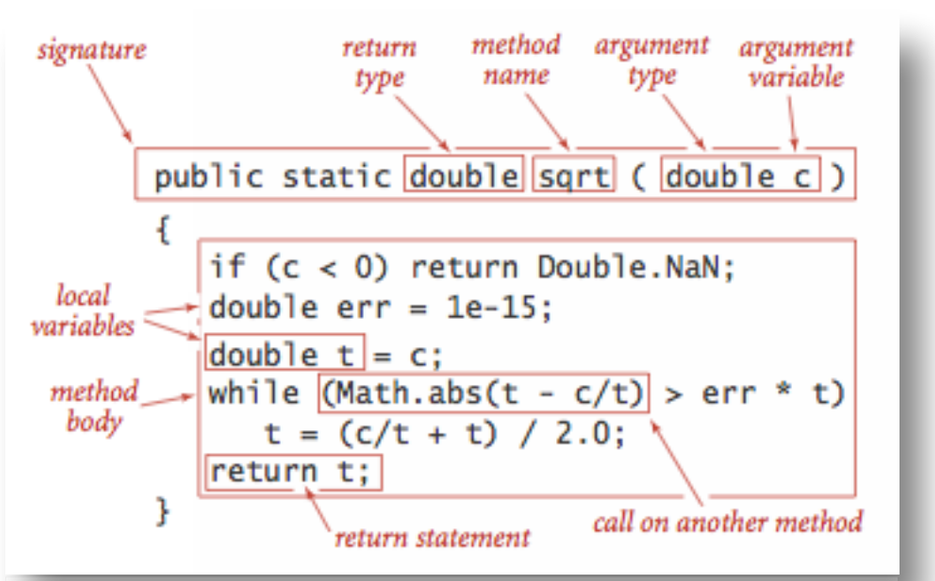
1.1 Programming Model

- Primitive data types and expressions
- Statements
- Static Methods

Static Methods

Static Methods - Static methods are called functions in many programming languages, since they can behave like mathematical functions. Each static method is a sequence of statements that are executed, one after the other, when the static method is called..

- Defining a static method. A method encapsulates a computation that is defined as a sequence of statements. A method takes arguments (values of given data types) and computes a return value of some data type or causes a side effect. Each static method is composed of a signature and a body..



- Invoking a static method. A call on a static method is its name followed by expressions that specify argument values in parentheses, separated by commas. When a method is called, its argument variables are initialized with the values of the corresponding expressions in the call. A return statement terminates a static method, returning control to the caller. If the static method is to compute a value, that value must be specified in a return statement

Static Methods

- *Properties of methods.* Java methods have the following features:

  - Arguments are passed by value. When calling a function, the argument value is fully evaluated and the resulting value is copied into argument variable. This is known as pass by value. Array (and other object) references are also passed by value: the method cannot change the reference, but it can change the entries in the array (or value of the object).
  - Method names can be overloaded. Methods within a class can have the same name, provided they have different signatures. This features is known as overloading.
  - A method has a single return value but may have multiple return statements. A Java method can provide only one return value. Control goes back to the calling program as soon as the first return statement is reached.
  - A method can have side effects. A method may use the keyword void as its return type, to indicate that it has no return value and produces side effects (consume input, produce output, change entries in an array, or otherwise change the state of the system).

- *Recursion. A recursive method is a method that calls itself either directly or indirectly. There are three important rules of thumb in developing recursive programs:*

  - The recursion has a base case.
  - Recursive calls must address subproblems that are smaller in some sense, so that recursive calls converge to the base case.
  - Recursive calls should not address subproblems that overlap.

- Basic programming model. A library of static methods is a set of static methods that are defined in a Java class. A basic model for Java programming is to develop a program that addresses a specific computational task by creating a library of static methods, one of which is named main().

Static Methods

- *Modular programming. Libraries of static methods enable modular programming, where static methods in one library can call static methods defined in other libraries. This approach has many important advantages.*

  - Work with modules of reasonable size
  - Share and reuse code without having to reimplement it
  - Substitute improved implementations
  - Develop appropriate abstract models for addressing programming problems
  - Localize debugging

- *External libraries. We use static methods from four different kinds of libraries, each requiring (slightly) differing procedures for code reuse.*

  - Standard system libraries in java.lang, including java.lang.Math, java.lang.Integer, and java.lang.Double.
  - Imported system libraries such as java.util.Arrays. An import statement at the beginning of the program is needed to use such libraries.
  - Other libraries in this book. To use such a program, download the source from the booksite into your working directory or follow these instructions for adding algs4.jar to your classpath.
  - The standard libraries that we have developed for use in this book. To use such a program, download the source from the booksite into your working directory or follow these instructions for adding stdlib.jar to your classpath.

To invoke a method from another library, we prepend the library name to the method name for each call: Math.sqrt(), Arrays.sort(), BinarySearch.rank(), and StdIn.readInt().

- Unit testing. A best practice in Java programming is to include a main() in every library of static methods that tests the methods in the library.

Static Methods

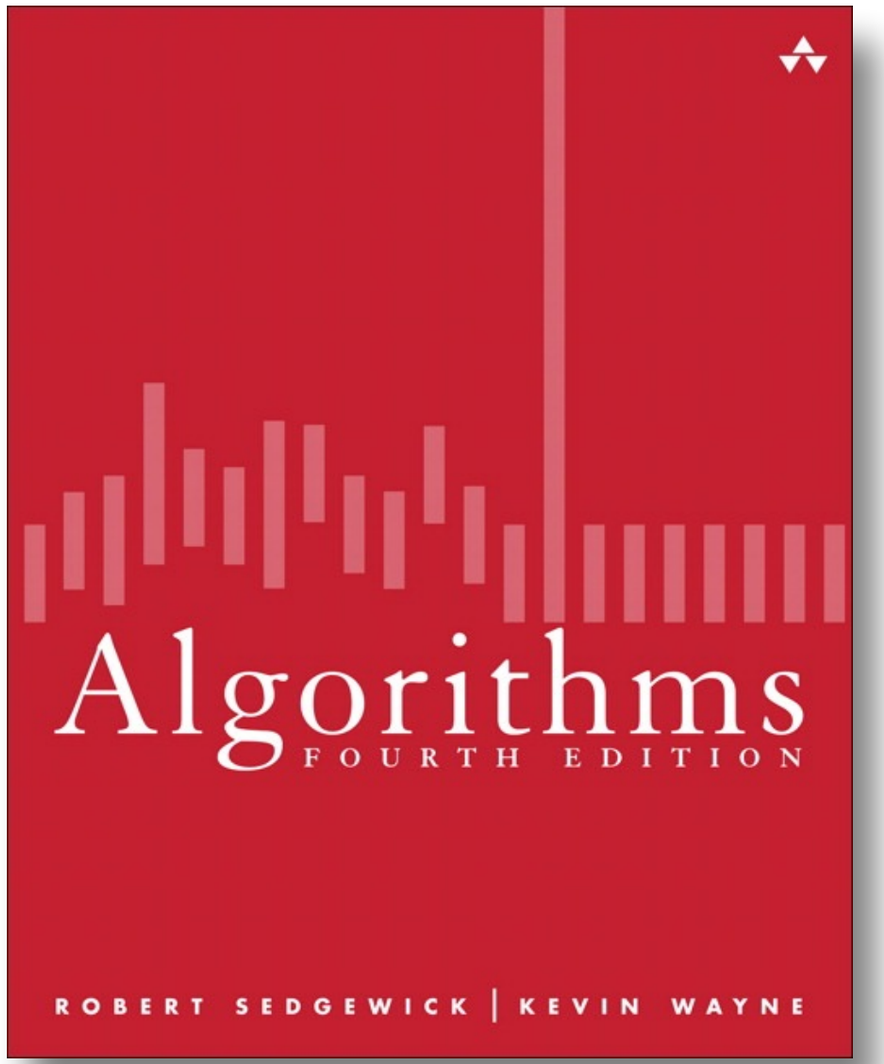
- *Strings.*

type	set of values	typical literals	operators	typical expressions	
				expression	value
String	character sequences	"AB" "Hello" "2.5"	+ (concatenate)	"Hi, " + "Bob" "12" + "34" "1" + "+" + "2"	"Hi, Bob" "1234" "1+2"

- *Formatted output.*

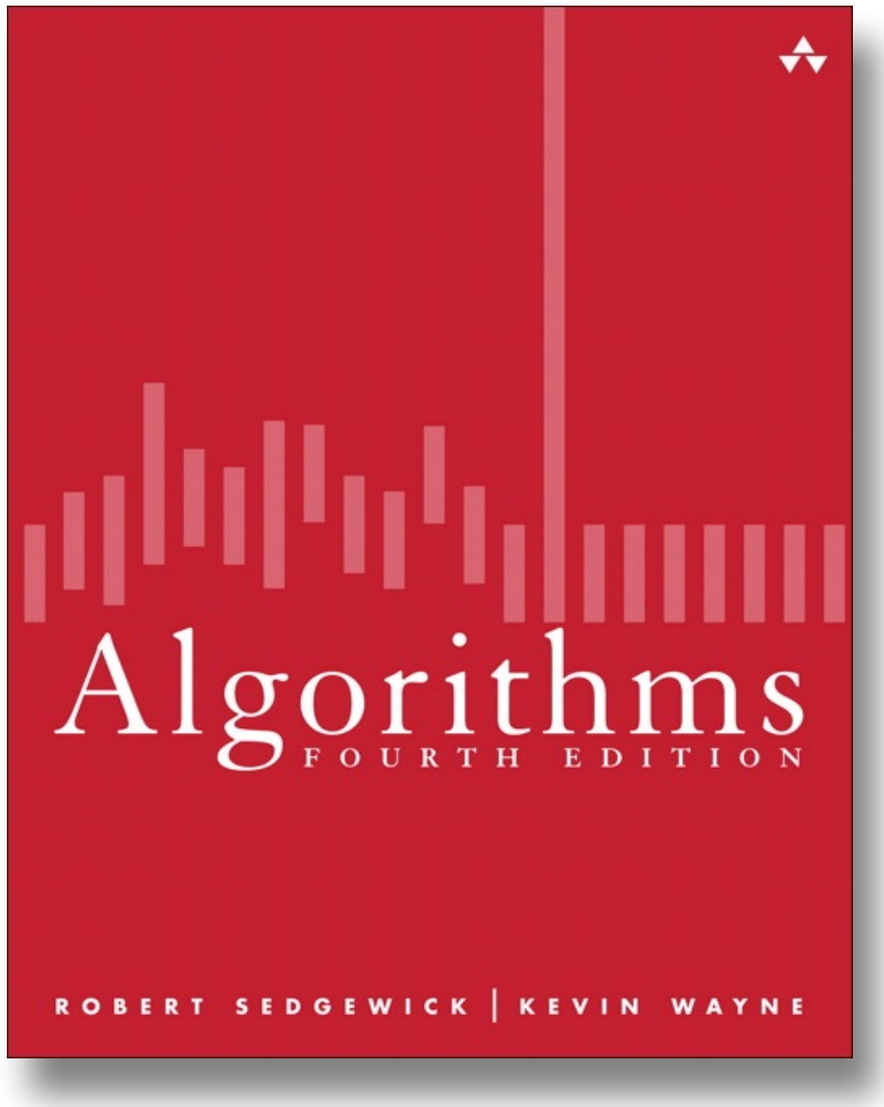
type	code	typical literal	sample format strings	converted string values for output
int	d	512	"%14d" "%-14d"	" 512" "512 "
double	f e	1595.1680010754388	"%14.2f" "%.7f" "%14.4e"	" 1595.17" "1595.1680011" " 1.5952e+03"
String	s	"Hello, World"	"%14s" "%-14s" "%-14.5s"	" Hello, World" "Hello, World " "Hello "

Lecture Overview



1.2 Data Abstraction

- Object-oriented programming
- Using abstract data types
- Examples of abstract data types
- Implementing abstract data types
- Designing abstract data types



1.2 Data Abstraction

- Object-oriented programming
- Using abstract data types
- Examples of abstract data types
- Implementing abstract data types
- Designing abstract data types

Object-oriented Programming

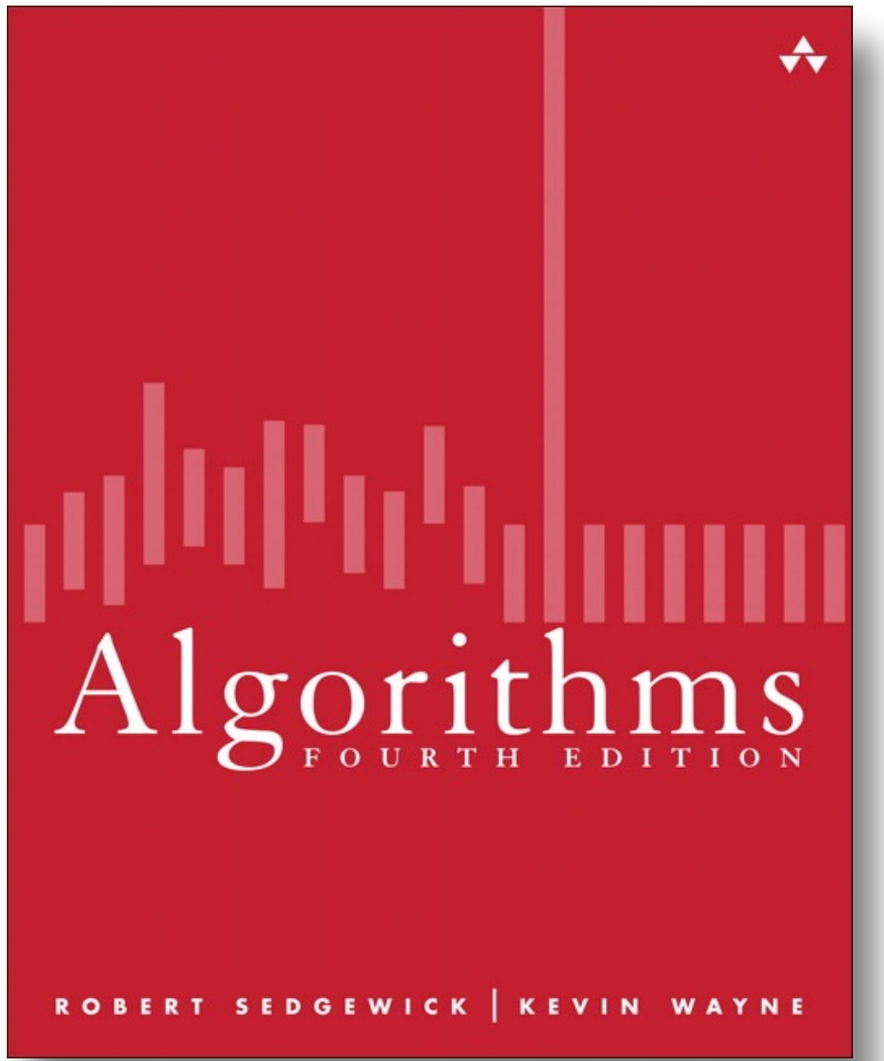
- *Programming in Java is largely based on building data types. This style of programming is known as object-oriented programming, as it revolves around the concept of an object, an entity that holds a data type value. With Java's primitive types we are largely confined to programs that operate on numbers, but with reference types we can write programs that operate on strings, pictures, sounds, 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 the range of data 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.*



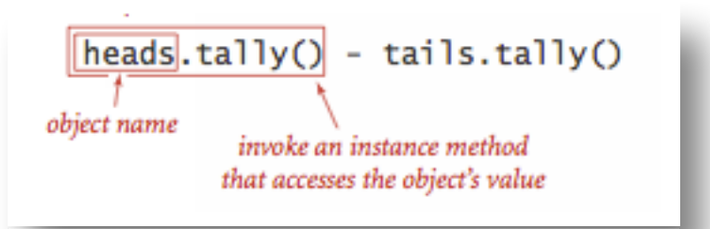
1.2 Data Abstraction

- Object-oriented programming
- Using abstract data types
- Examples of abstract data types
- Implementing abstract data types
- Designing abstract data types

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 Abstract Data Types

- *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.

- *Assignment statements.* An assignment statement with a reference type creates a copy of the reference (and 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: (The code prints the string "2 ones".)

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

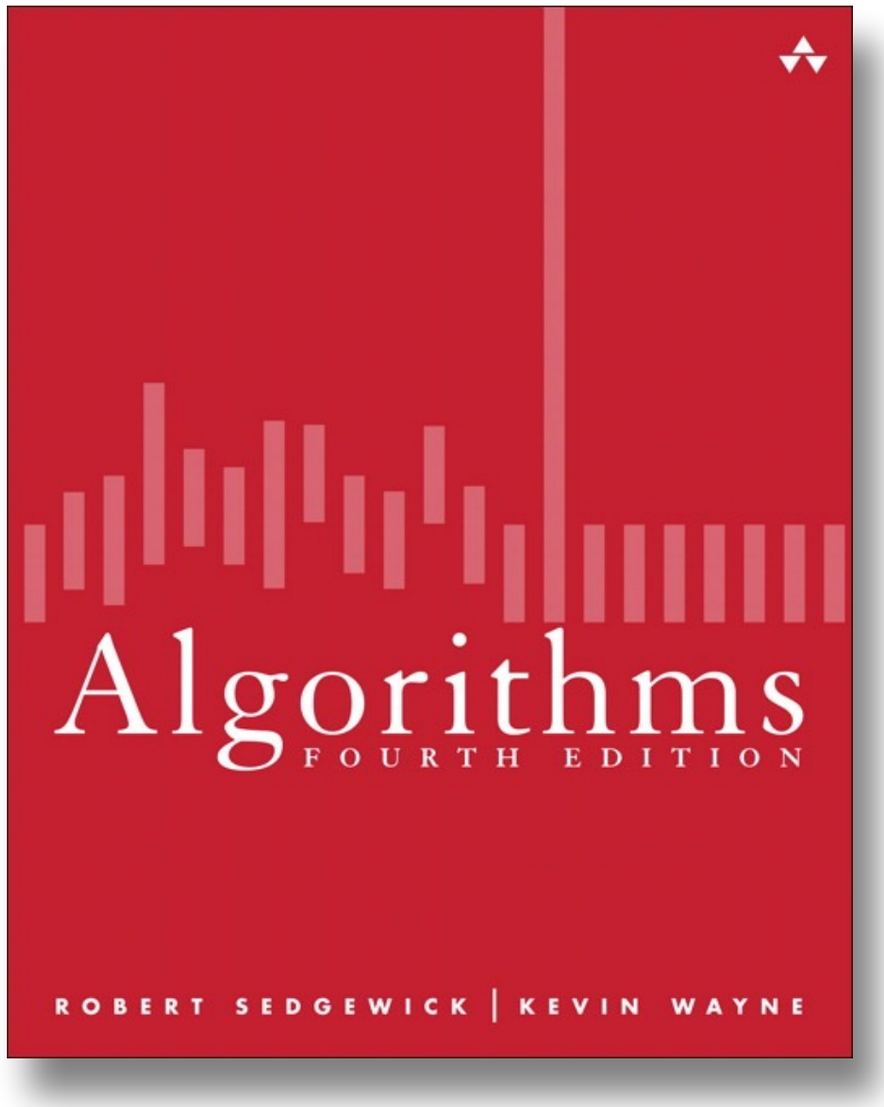
- *Objects as arguments. You can pass objects as arguments to methods. Java passes a copy of the argument value from the calling program to the method. This arrangement is known as pass by value. If you pass a reference to an object of type Counter, Java passes a copy of that reference. Thus, the method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

Using Abstract Data Types

- *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.*



1.2 Data Abstraction

- Object-oriented programming
- Using abstract data types
- Examples of abstract data types
- Implementing abstract data types
- Designing abstract data types

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)*

Examples of Abstract Data Types

- *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. 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*

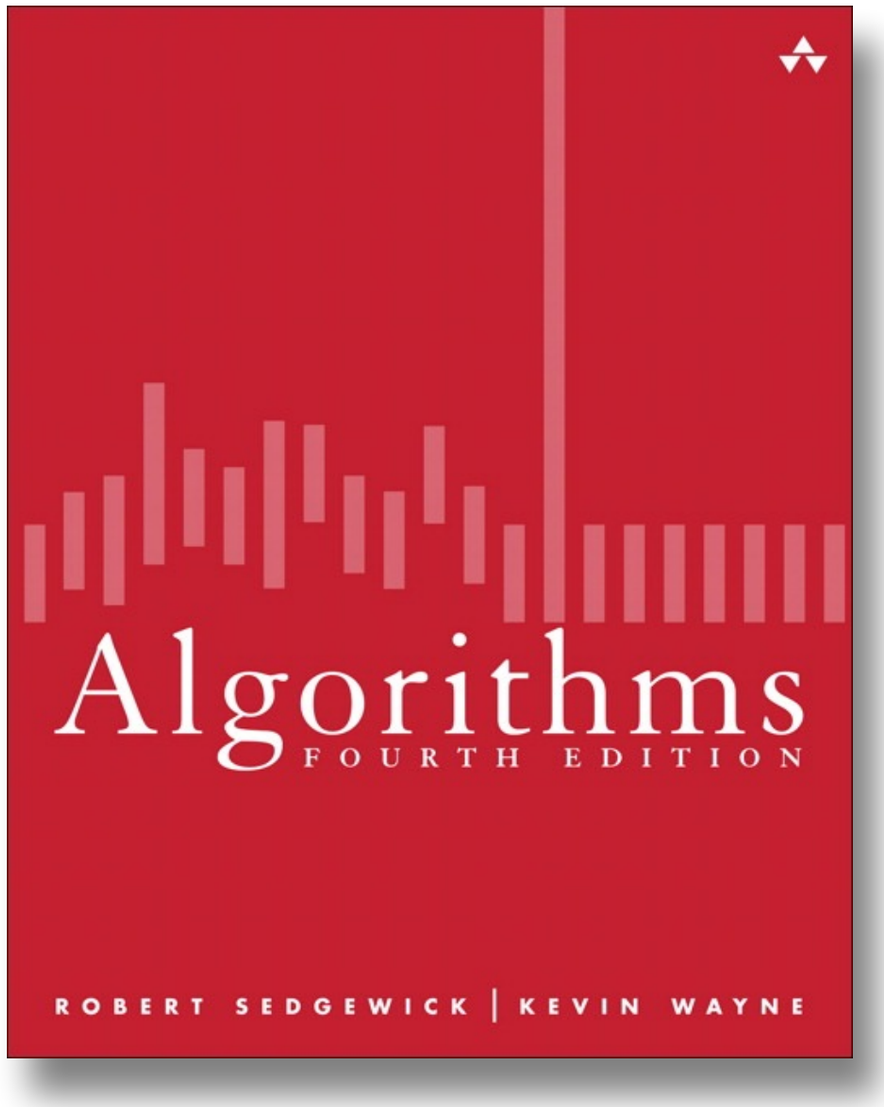
public class String

	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)

Examples of Abstract Data Types

- *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 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 library includes the data types In.java, Out.java, and Draw.java that support multiple input and output streams*



1.2 Data Abstraction

- Object-oriented programming
- Using abstract data types
- Examples of abstract data types
- Implementing abstract data types
- Designing abstract data types

Implementing Abstract Data Types

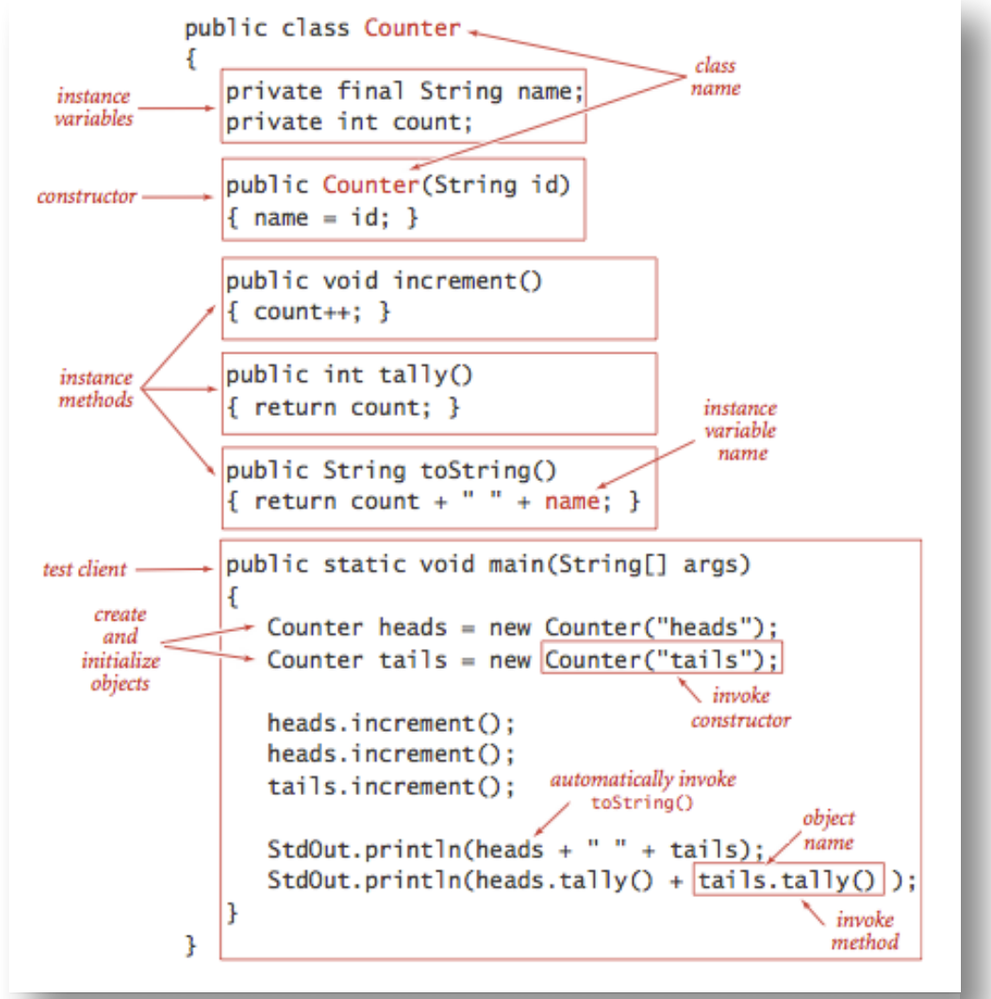
- We implement ADTs with a Java class, putting the code in a file with the same name as the class, followed by the .java extension. The first statements in the file declare *instance variables* that define the data-type values. Following the instance variables are the *constructor* and the *instance 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*

Implementing Abstract Data Types

- 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 client values, the statements are executed until a return value is computed, and the value is 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).



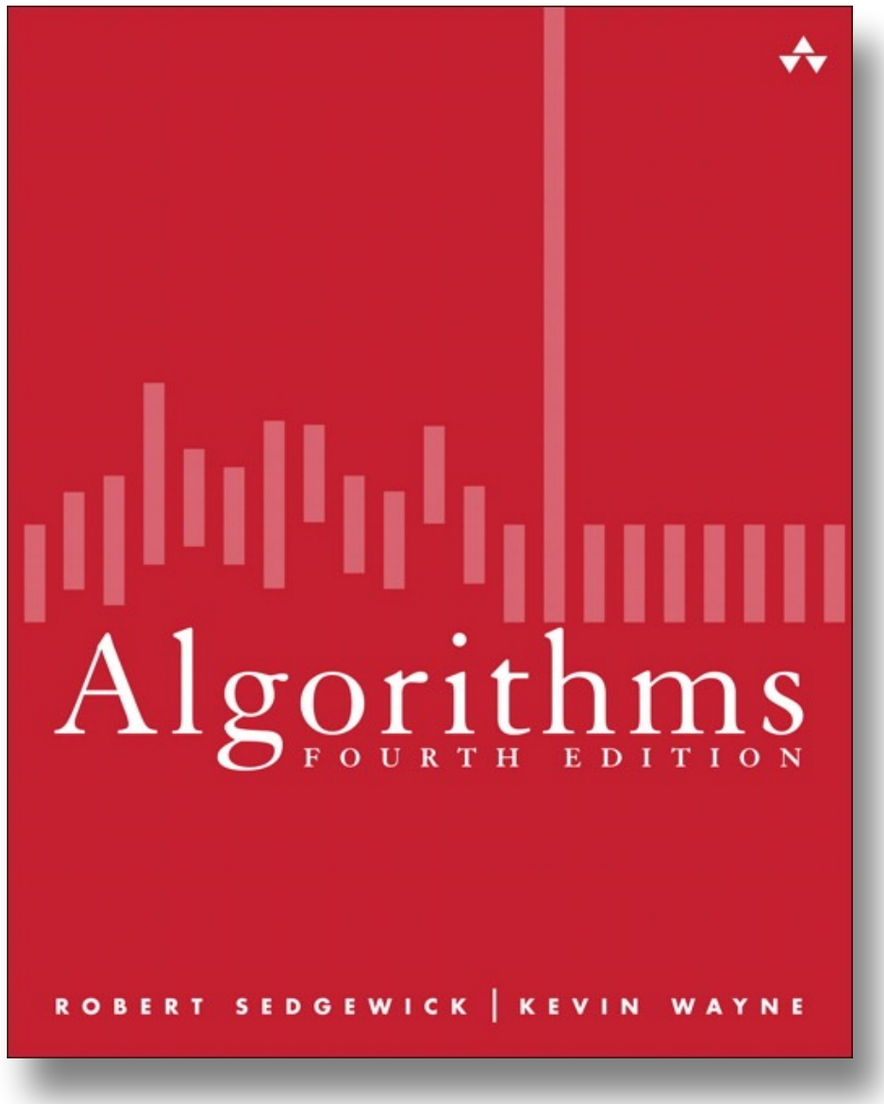
Implementing Abstract Data Types

- 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).

```
public class Example                    instance variable
{
   private int var;
   ...

   private void method1()
   {
      int var;                          local variable
      ...  var     ...                  refers to local variable, NOT instance variable
      ...  this.var ...                 refers to instance variable
   }

   private void method2()
   {
      ...  var     ...                  refers to instance variable
   }
   ...
}
```



1.2 Data Abstraction

- Object-oriented programming
- Using abstract data types
- Examples of abstract data types
- Implementing abstract data types
- Designing abstract data types

Designing Abstract Data Types

- *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 diffuse 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*

Designing Abstract Data Types

- *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.*

- *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 in 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

Designing Abstract Data Types

- *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 without 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 it 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

Designing Abstract Data Types

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

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

Designing Abstract Data Types

- *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. (Adhering to these Java conventions can be tricky, as illustrated for Date.java and Transaction.java.)*
  - *Consistent: multiple invocations of x.equals(y) consistently return the same value, provided neither object is modified.*
  - *Not null: x.equals(null) returns false.*

Designing Abstract Data Types

- *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.

Designing Abstract Data Types

- *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 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;
```

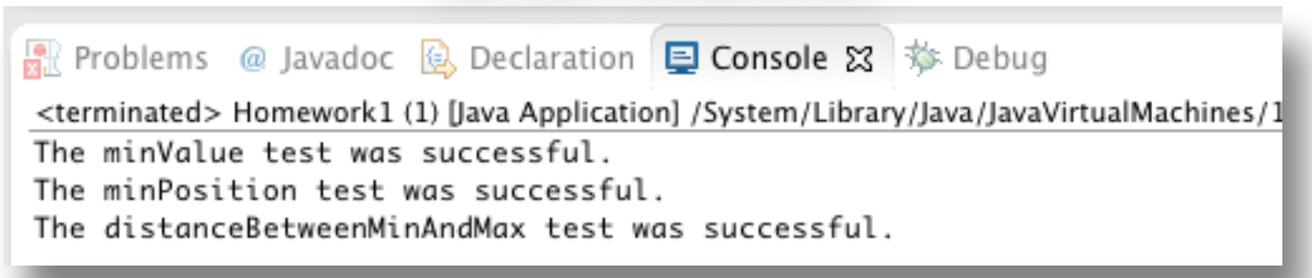
- *you can pinpoint the place where the error occurred. By default, assertions are disabled.You can enable them from the command line by using the -enableassertions flag (-ea for short). Assertions are for debugging: your program should not rely on assertions for normal operation since they may be disabled.*

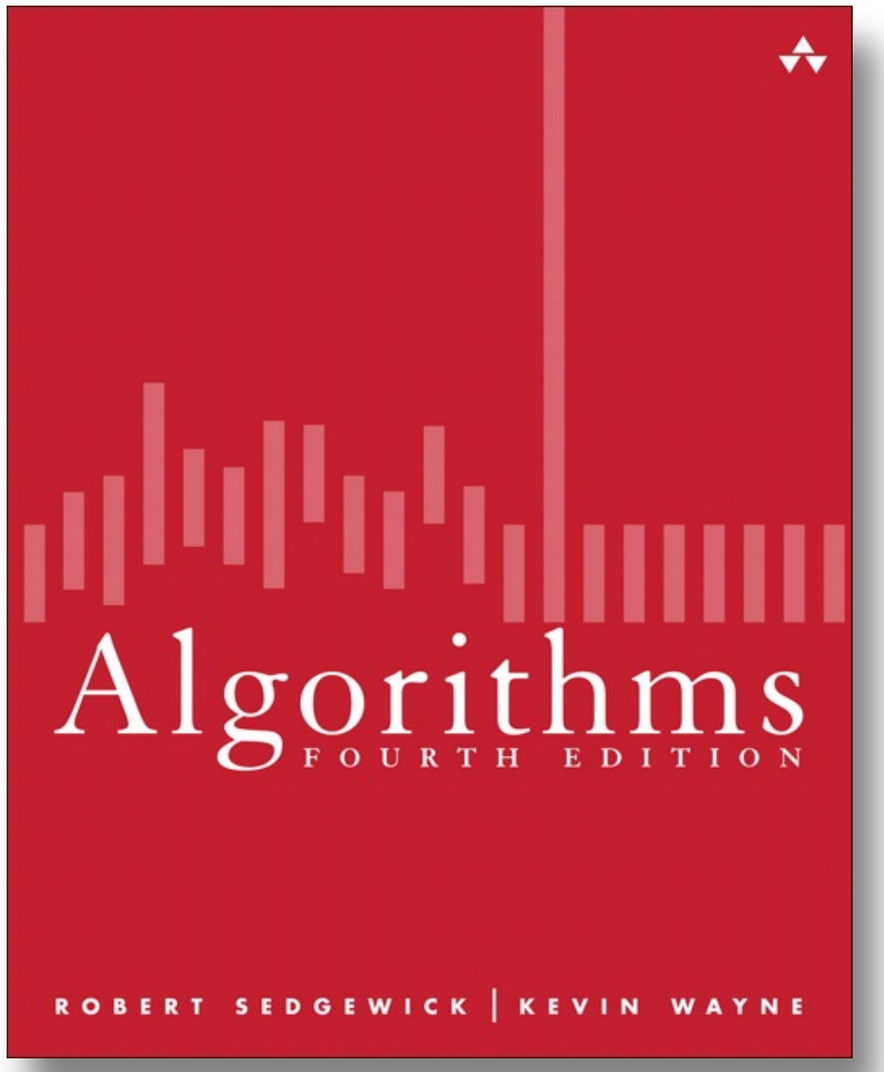
Assignment

Modify the Homework1.java application to provide valid logic for the following methods:

- minValue
- minPosition
- distanceBetweenMinAndMax

Sample Screenshot:





Assignment due next Monday at 11:59 PM