BIG JAVA

Javadoc Commenting

```
Provide documentation comments for
every class, every method,
every parameter variable, and every return value.
```

You can invoke the *javadoc* utility from a shell window, by issuing the command:

```
javadoc MyClass.java
```

or, if you want to document multiple Java files,

```
javadoc *.java
```

The javadoc utility copies the first sentence of each comment to a summary table in the HTML documentation.
Therefore, it is best to write that first sentence with some care.
It should start with an uppercase letter and end with a period.

```
Starts with /** and Ends with */

/**
Withdraws money from the bank account.
@param amount the amount to withdraw
*/
public void withdraw(double amount)
{
implementation—filled in later
}
/**
Gets the current balance of the bank account.
@return the current balance
*/
public double getBalance()
{
implementation—filled in later
}
```

Unit Testing

```
To test a class, use an environment for interactive testing,
or write a tester class to execute test instructions.
```

Alternatively, you can write a tester class. A tester class is a class with a main method that contains statements to run methods of another class. As discussed in Section 2.7, a tester class typically carries out the following steps:

1. Construct one or more objects of the class that is being tested.
2. Invoke one or more methods.
3. Print out one or more results.
4. Print the expected results.

Primitive Types

```
Java has eight primitive types, including four integer types
and two floatingpoint types.
```

- Notes: The largest number that can be represented in an int is denoted by *Integer.MAX_VALUE*. Its value is about 2.14 billion. Similarly, the smallest integer is *Integer.MIN_VALUE*, about –2.14 billion.

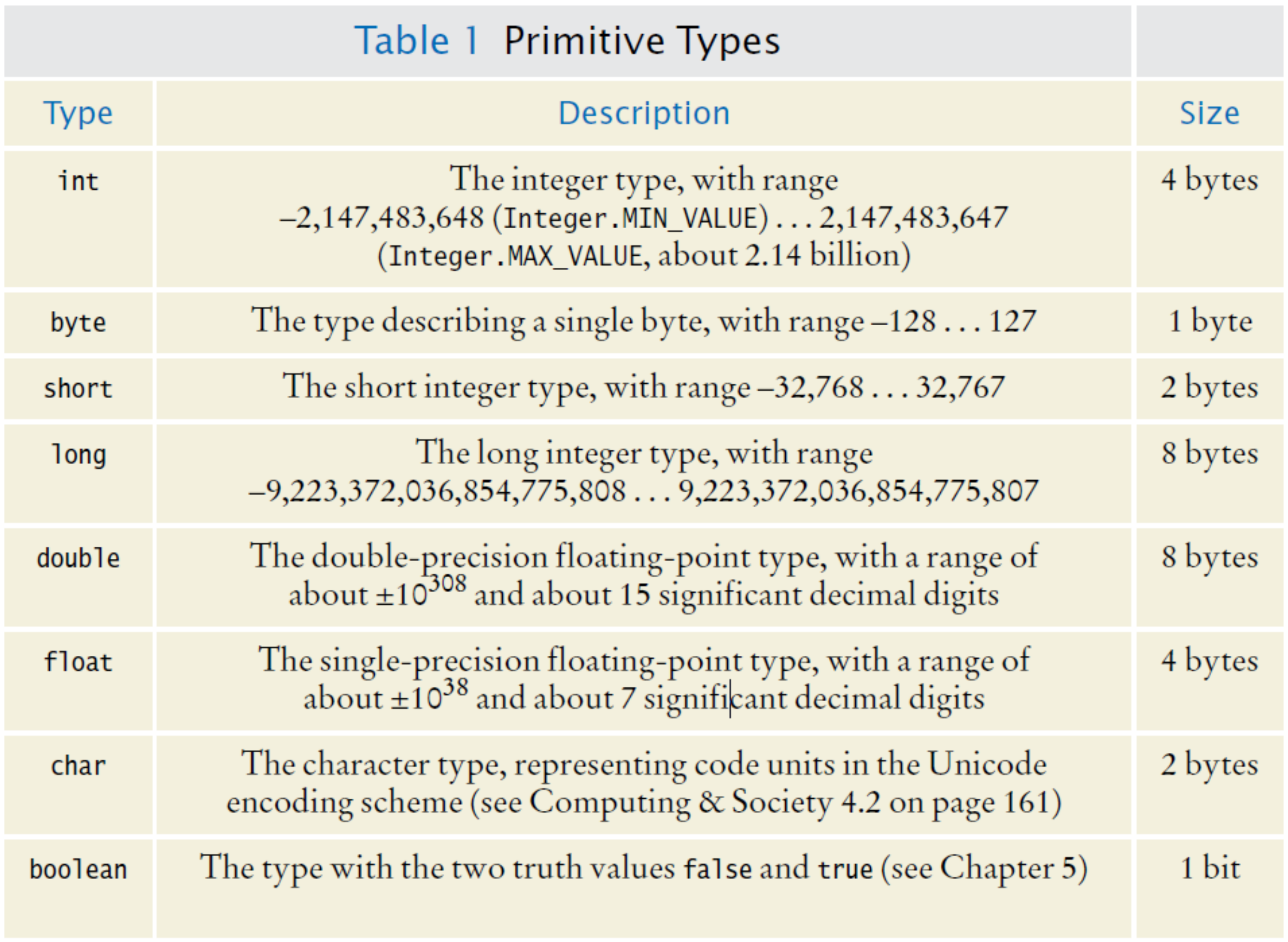


Table 1 Primitive Types

Type	Description	Size
`int`	The integer type, with range –2,147,483,648 (`Integer.MIN_VALUE`) . . . 2,147,483,647 (`Integer.MAX_VALUE`, about 2.14 billion)	4 bytes
`byte`	The type describing a single byte, with range –128 . . . 127	1 byte
`short`	The short integer type, with range –32,768 . . . 32,767	2 bytes
`long`	The long integer type, with range –9,223,372,036,854,775,808 . . . 9,223,372,036,854,775,807	8 bytes
`double`	The double-precision floating-point type, with a range of about $\pm 10^{308}$ and about 15 significant decimal digits	8 bytes
`float`	The single-precision floating-point type, with a range of about $\pm 10^{38}$ and about 7 significant decimal digits	4 bytes
`char`	The character type, representing code units in the Unicode encoding scheme (see Computing & Society 4.2 on page 161)	2 bytes
`boolean`	The type with the two truth values `false` and `true` (see Chapter 5)	1 bit

Keyword *final*

A *final* variable is a constant. Once its value has been set, it cannot be changed. Many programmers use all-uppercase names for constants (final variables), such as NICKEL_VALUE.

Frequently, constant values are needed in several methods. Then you should declare them together with the instance variables of a class and tag them as *static* and *final*.
As before, *final* indicates that the value is a constant. The *static* reserved word means that the constant belongs to the class — this is explained in greater detail in Chapter 8.):

```
public class CashRegister
    {
        // Constants
        public static final double QUARTER_VALUE = 0.25;
        public static final double DIME_VALUE = 0.1;
        public static final double NICKEL_VALUE = 0.05;
        public static final double PENNY_VALUE = 0.01;

        // Instance variables
        private double purchase;
        private double payment;

        // Methods
        . . .
    }
```

Big Numbers

If you want to compute with really large numbers, you can use big number objects. Big number objects are objects of the *BigInteger* and *BigDecimal* classes in the java.math package. Unlike the number types such as int or double, big number objects have essentially no limits on their size and precision. However, computations with big number objects are much slower than those that involve number types. Perhaps more importantly, you can't use the familiar arithmetic operators such as (+ - *) with them. Instead, you have to use methods called add, subtract, and multiply. Here is an example of how to create a *BigInteger* object and how to call the multiply method:

```
BigInteger n = new BigInteger("1000000");
BigInteger r = n.multiply(n);
System.out.println(r); // Prints 1000000000000
```

The *BigDecimal* type carries out floating-point computations without roundoff errors. For example,

```
BigDecimal d = new BigDecimal("4.35");
BigDecimal e = new BigDecimal("100");
BigDecimal f = d.multiply(e);
System.out.println(f); // Prints 435.00
```

Arithmetics

Table 4 Mathematical Methods

Method	Returns	Method	Returns
`Math.sqrt(x)`	Square root of x (≥ 0)	`Math.abs(x)`	Absolute value $\lvert x \rvert$
`Math.pow(x, y)`	x^y ($x > 0$, or $x = 0$ and $y > 0$, or $x < 0$ and y is an integer)	`Math.max(x, y)`	The larger of x and y
`Math.sin(x)`	Sine of x (x in radians)	`Math.min(x, y)`	The smaller of x and y
`Math.cos(x)`	Cosine of x	`Math.exp(x)`	e^x
`Math.tan(x)`	Tangent of x	`Math.log(x)`	Natural log ($\ln(x)$, $x > 0$)
`Math.round(x)`	Closest integer to x (as a `long`)	`Math.log10(x)`	Decimal log ($\log_{10}(x)$, $x > 0$)
`Math.ceil(x)`	Smallest integer $\geq x$ (as a `double`)	`Math.floor(x)`	Largest integer $\leq x$ (as a `double`)
`Math.toRadians(x)`	Convert x degrees to radians (i.e., returns $x \cdot \pi / 180$)	`Math.toDegrees(x)`	Convert x radians to degrees (i.e., returns $x \cdot 180 / \pi$)

For converting a value to another datatype use a cast *(typename)*:

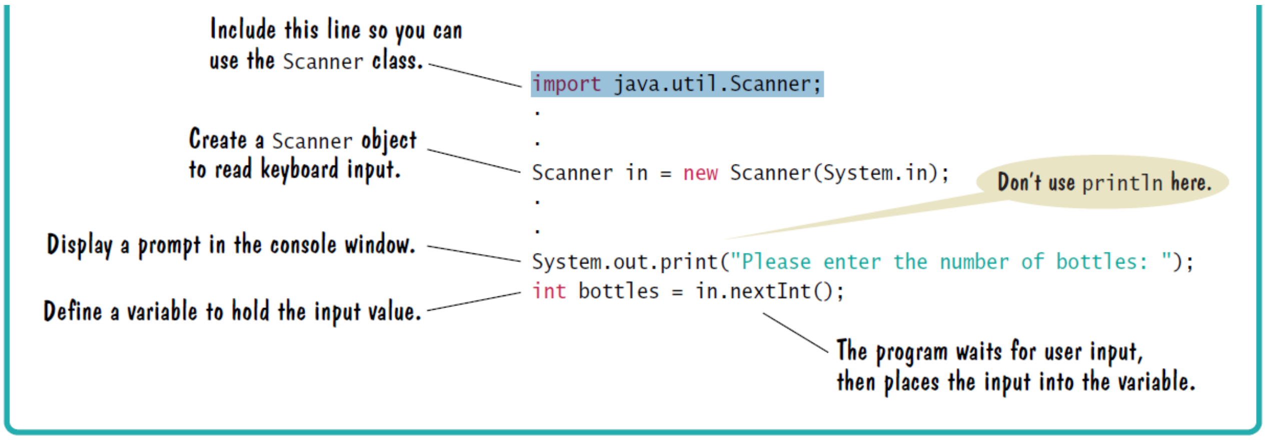
```
int dollars = (int) (total + tax);
```

To round it:

```
long rounded = Math.round(balance);
```

Input an Output

Syntax 4.3 Input Statement


Include this line so you can use the Scanner class.
Create a Scanner object to read keyboard input.
Don't use println here.
Display a prompt in the console window.
Define a variable to hold the input value.
The program waits for user input, then places the input into the variable.

```
import java.util.Scanner;
.
.
Scanner in = new Scanner(System.in);
.
.
System.out.print("Please enter the number of bottles: ");
int bottles = in.nextInt();
```

To format output we use:

```
System.out.printf("Quantity: %d Total: %10.2f", quantity, total);
```

The *printf* method with options:

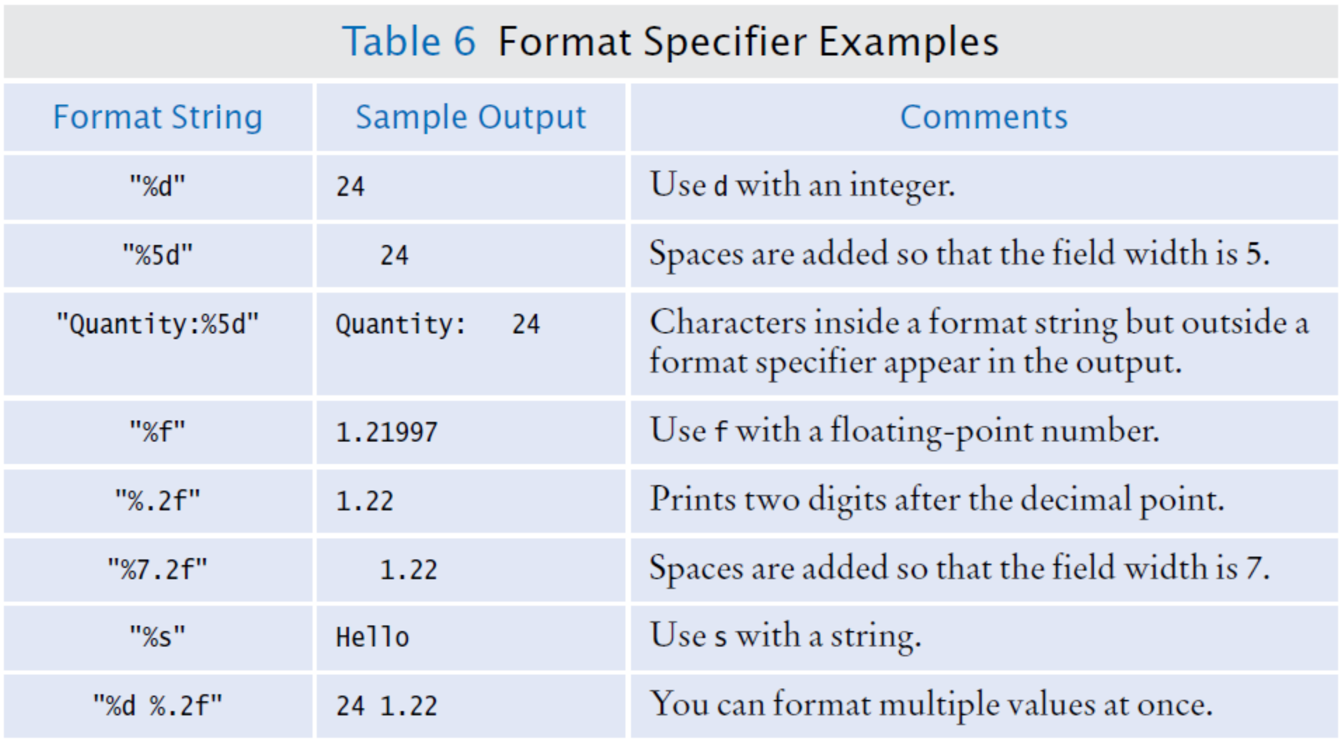


Table 6 Format Specifier Examples

Format String	Sample Output	Comments
"%d"	`24`	Use d with an integer.
"%5d"	`   24`	Spaces are added so that the field width is 5.
"Quantity:%5d"	`Quantity:   24`	Characters inside a format string but outside a format specifier appear in the output.
"%f"	`1.21997`	Use f with a floating-point number.
"%.2f"	`1.22`	Prints two digits after the decimal point.
"%7.2f"	`   1.22`	Spaces are added so that the field width is 7.
"%s"	`Hello`	Use s with a string.
"%d %.2f"	`24 1.22`	You can format multiple values at once.

Decisions

Java has a conditional operator of the form

```
condition ? value1 : value2
```

The value of that expression is either value1 if the test passes or value2 if it fails. For example, we can compute the actual floor number as

```
actualFloor = floor > 13 ? floor - 1 : floor;
```

which is equivalent to

```
if (floor > 13) { actualFloor = floor - 1; } else { actualFloor = floor; }
```

Enumeration Types

An enumeration type has a finite set of values, for example

```
public enum FilingStatus { SINGLE, MARRIED, MARRIED_FILING_SEPARATELY }
```

You can have any number of values, but you must include them all in the enum declaration.
You can declare variables of the enumeration type:

```
FilingStatus status = FilingStatus.SINGLE;
```

If you try to assign a value that isn't a FilingStatus, such as 2 or "S", then the compiler reports an error.
Use the == operator to compare enumeration values, for example:

```
if (status == FilingStatus.SINGLE) . . .
```

Place the enum declaration inside the class that implements your program, such as

```
public class TaxReturn
{
    public enum FilingStatus { SINGLE, MARRIED, MARRIED_FILING_SEPARATELY }
    . . .
}
```

Printing Logs

Instead of printing directly to System.out, use the global logger object that is returned by the call `Logger.getGlobal()`. Then call the info method:

```
Logger.getGlobal().info("status is SINGLE");
```

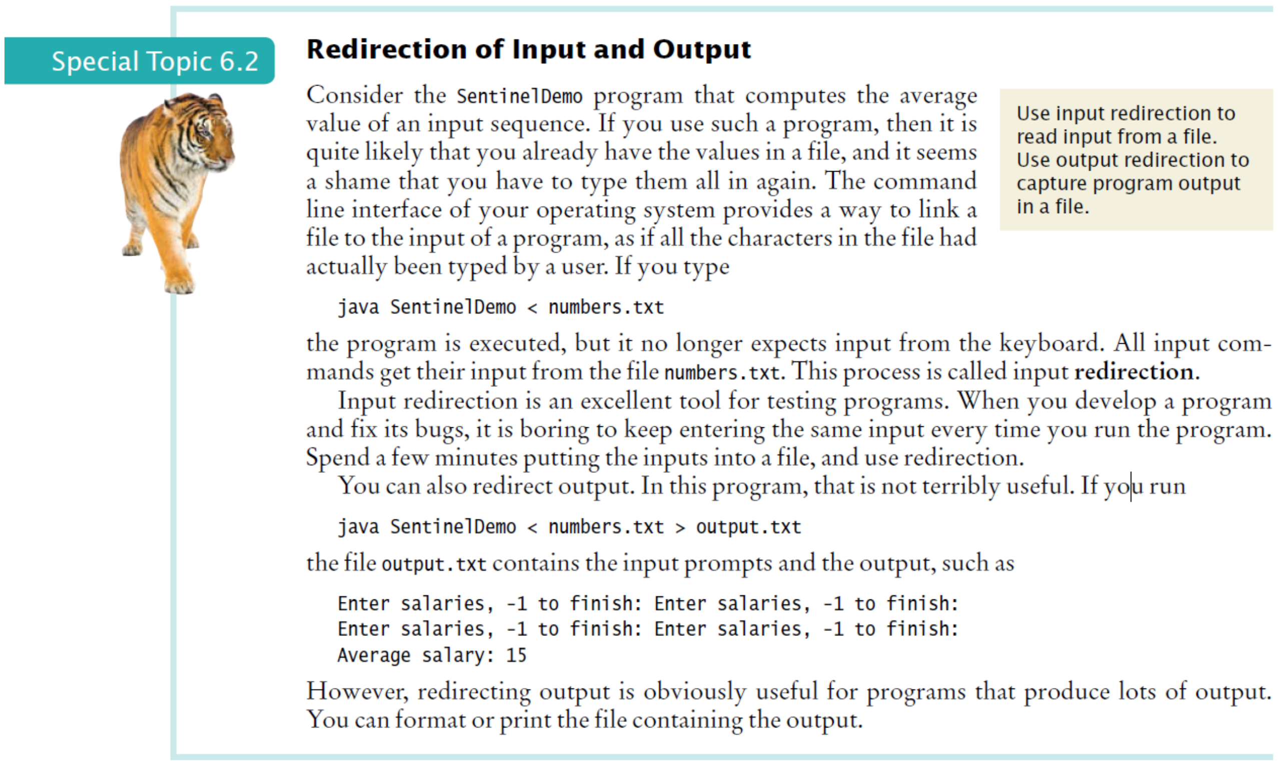
By default, the message is printed. But if you call

```
Logger.getGlobal().setLevel(Level.OFF);
```

at the beginning of the main method of your program, all log message printing is suppressed.
Set the level to `Level.INFO` to turn logging of info messages on again.

Thus, you can turn off the log messages when your program works fine, and you can turn them back on if you find another error. In other words, using `Logger.getGlobal().info` is just like `System.out.println`, except that you can easily activate and deactivate the logging.

Redirect Input and Output



Special Topic 6.2

Redirection of Input and Output

Consider the `SentinelDemo` program that computes the average value of an input sequence. If you use such a program, then it is quite likely that you already have the values in a file, and it seems a shame that you have to type them all in again. The command line interface of your operating system provides a way to link a file to the input of a program, as if all the characters in the file had actually been typed by a user. If you type

Use input redirection to read input from a file. Use output redirection to capture program output in a file.

```
java SentinelDemo < numbers.txt
```

the program is executed, but it no longer expects input from the keyboard. All input commands get their input from the file `numbers.txt`. This process is called input **redirection**.

Input redirection is an excellent tool for testing programs. When you develop a program and fix its bugs, it is boring to keep entering the same input every time you run the program. Spend a few minutes putting the inputs into a file, and use redirection.

You can also redirect output. In this program, that is not terribly useful. If you run

```
java SentinelDemo < numbers.txt > output.txt
```

the file `output.txt` contains the input prompts and the output, such as

```
Enter salaries, -1 to finish: Enter salaries, -1 to finish:
Enter salaries, -1 to finish: Enter salaries, -1 to finish:
Average salary: 15
```

However, redirecting output is obviously useful for programs that produce lots of output. You can format or print the file containing the output.

Generating Random Numbers

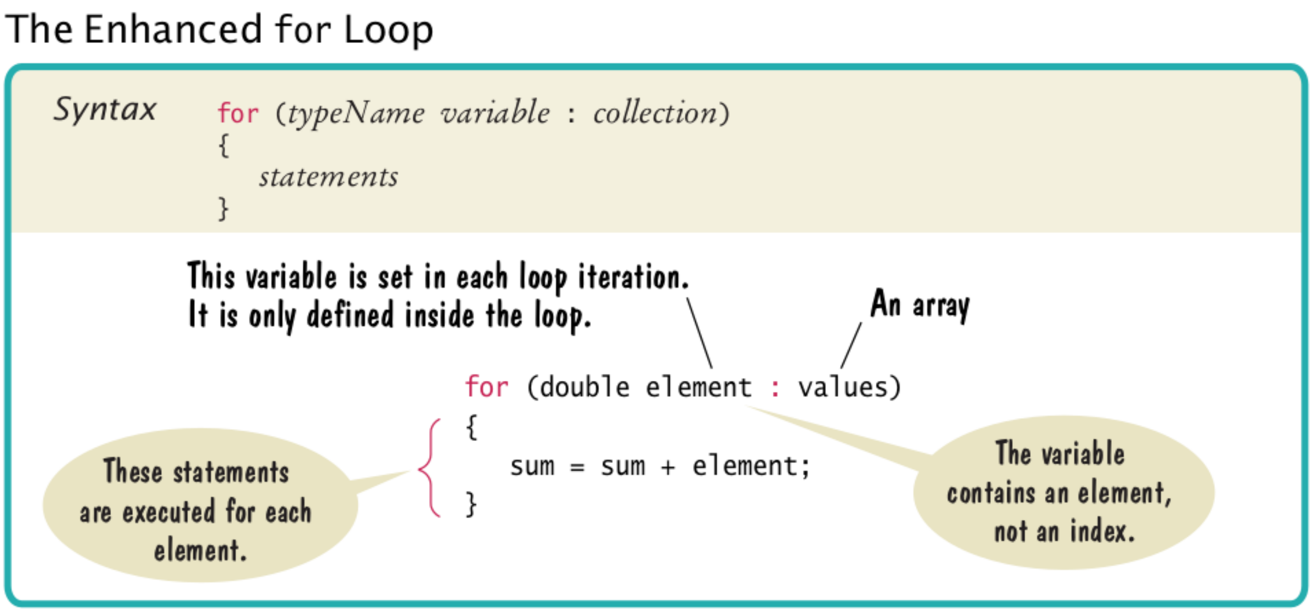
Method	Returns
nextInt(n)	A random integer between the integers 0 (inclusive) and n (exclusive)
nextDouble()	A random floating-point number between 0 (inclusive) and 1 (exclusive)

```
Random generator = new Random();
int d = 1 + generator.nextInt(6);
```

Enhanced For Loop

Use the enhanced for loop if you do not need the index values in the loop body.

```
for (double element : values)
{
    ...
    element = 0; // ERROR: this assignment does not modify array elements
}
```


The Enhanced for Loop
Syntax
for (typeName variable : collection)
{
statements
}
This variable is set in each loop iteration. It is only defined inside the loop.
An array
for (double element : values)
{
sum = sum + element;
}
These statements are executed for each element.
The variable contains an element, not an index.

Arrays as Parameters

It is possible to declare methods that receive a variable number of arguments. For example, we can write a method that can add an arbitrary number of scores to a student:

```
fred.addScores(10, 7); // This method call has two arguments
fred.addScores(1, 7, 2, 9); // Another call to the same method, now with four arguments
```

The method must be declared as

```
public void addScores(int... values)
```

The `int...` type indicates that the method can receive any number of int arguments. The values parameter variable is actually an `int[] array` that contains all arguments that were passed to the method

Array Lists

Table 2 Working with Array Lists

`ArrayList<String> names = new ArrayList<String>();`	Constructs an empty array list that can hold strings.

`names.add("Ann");`
`names.add("Cindy");`
	Adds elements to the end of the array list.

`System.out.println(names);`	Prints `[Ann, Cindy]`.
`names.add(1, "Bob");`	Inserts an element at index 1. `names` is now `[Ann, Bob, Cindy]`.
`names.remove(0);`	Removes the element at index `0`. `names` is now `[Bob, Cindy]`.
`names.set(0, "Bill");`	Replaces an element with a different value. `names` is now `[Bill, Cindy]`.
`String name = names.get(i);`	Gets an element.
`String last = names.get(names.size() - 1);`	Gets the last element.

`ArrayList<Integer> squares = new ArrayList<Integer>();`
`for (int i = 0; i < 10; i++)`
`{`
`   squares.add(i * i);`
`}`
	Constructs an array list holding the first ten squares.

For most programming tasks, array lists are easier to use than arrays. Array lists can grow and shrink. On the other hand, arrays have a nicer syntax for element access and initialization. Which of the two should you choose? Here are some recommendations.

- If the size of a collection never changes, use an array.
- If you collect a long sequence of primitive type values and you are concerned about efficiency, use an array.
- Otherwise, use an array list.

Designing Classes

A class should represent a single concept from a problem domain, such as business, science, or mathematics.

Very occasionally, a class has no objects, but it contains a collection of related static methods and constants. The Math class is an example. Such a class is called a utility class. Finally, you have seen classes with only a main method. Their sole purpose is to start a program. From a design perspective, these are somewhat degenerate examples of classes.

The public interface of a class is ***cohesive*** if all of its features are related to the concept that the class represents.

Some classes have been designed to have only accessor methods and no mutator methods at all. Such classes are called **immutable**. An example is the String class. Once a string has been constructed, its content never changes. No method in the String class can modify the contents of a string. For example, the toUpperCase method does not change characters from the original string. Instead, it constructs a new string that contains the uppercase characters.

As a rule of thumb, a method that returns a value should not be a mutator. For example, one would not expect that calling getBalance on a BankAccount object would change the balance. If you follow this rule, then all mutators of your class have return type void.

Sometimes, this rule is bent a bit, and mutator methods return an informational value. For example, the ArrayList class has a remove method to remove an object.

```
ArrayList<String> names = . . .;
boolean success = names.remove("Romeo");
```

That method returns true if the removal was successful; that is, if the list contained the object. Returning this value might be bad design if there was no other way to check whether an object exists in the list. However, there is such a method—the contains method. **It is acceptable for a mutator to return a value if there is also an accessor that computes it.**
The situation is less happy with the Scanner class. The next method is a mutator that returns a value. Unfortunately, there is no accessor that returns the same value. This sometimes makes it awkward to use a Scanner. You must carefully hang on to the value that the next method returns because you have no second chance to ask for it. It would have been better if there was another method, say *peek*, that yields the next input without consuming it.

Call by Value or by Reference

• A Java method can’t change the contents of any variable passed as an argument.
• A Java method can mutate an object when it receives a reference to it as an argument.

In Java, a method **can never change** the original contents of a **variable** that is passed to a method.

In Java, a method **can change** the state of **an object reference argument**, but it **cannot replace** the object reference with another.

Object Distinct States

If your object can have one of several states that affect the behavior, supply an instance variable for the current state.

```
public class Fish
{
    private int hungry;
    public static final int NOT_HUNGRY = 0;
    public static final int SOMEWHAT_HUNGRY = 1;
    public static final int VERY_HUNGRY = 2;
    . . .
}
(Alternatively, you can use an enumeration––see Special Topic 5.4)
```

Determine which methods change the state. In this example, a fish that has just eaten won't be hungry. But as the fish moves, it will get hungrier:

```
public void eat()
{
    hungry = NOT_HUNGRY;
    . . .
}
public void move()
{
    . . .
    if (hungry < VERY_HUNGRY) { hungry++; }
}
```

Finally, determine where the state affects behavior. A fish that is very hungry will want to look for food first:

```
public void move()
{
        if (hungry == VERY_HUNGRY)
        {
        Look for food.
        }
        . . .
}
```

Static Variables and Methods

A static variable belongs to the class, not to any object of the class.

```
public class BankAccount
{
    private double balance;
    private int accountNumber;
    private static int lastAssignedNumber = 1000;

    public BankAccount()
    {
       lastAssignedNumber++;
       accountNumber = lastAssignedNumber;
    }
    . . .
}
```

Every BankAccount object has its own balance and accountNumber instance variables, but **all objects share a single copy of the lastAssignedNumber variable**. That variable is stored in a separate location, outside any BankAccount objects.

Like instance variables, static variables should always be declared as private to ensure that methods of other classes do not change their values. However, static constants may be either private or public.

Sometimes a class defines methods that are not invoked on an object. Such a method is called a **static method**. A typical example of a static method is the *sqrt method in the Math clas*s. Because numbers aren't objects, you can't invoke methods on them. For example, if x is a number, then the call x.sqrt() is not legal in Java. Therefore, the Math class provides a static method that is invoked as Math.sqrt(x). No object of the Math class is constructed. The Math qualifier simply tells the compiler where to find the sqrt method.

There is a variant of the import directive that lets you use static methods and variables without class prefixes. For example,

```
import static java.lang.System.*;
import static java.lang.Math.*;
public class RootTester
{
    public static void main(String[] args)
    {
        double r = sqrt(PI); // Instead of Math.sqrt(Math.PI)
        out.println(r); // Instead of System.out
    }
}
```

Static imports can make programs easier to read, particularly if they use many mathematical functions.

Packages

A package is a set of related classes.

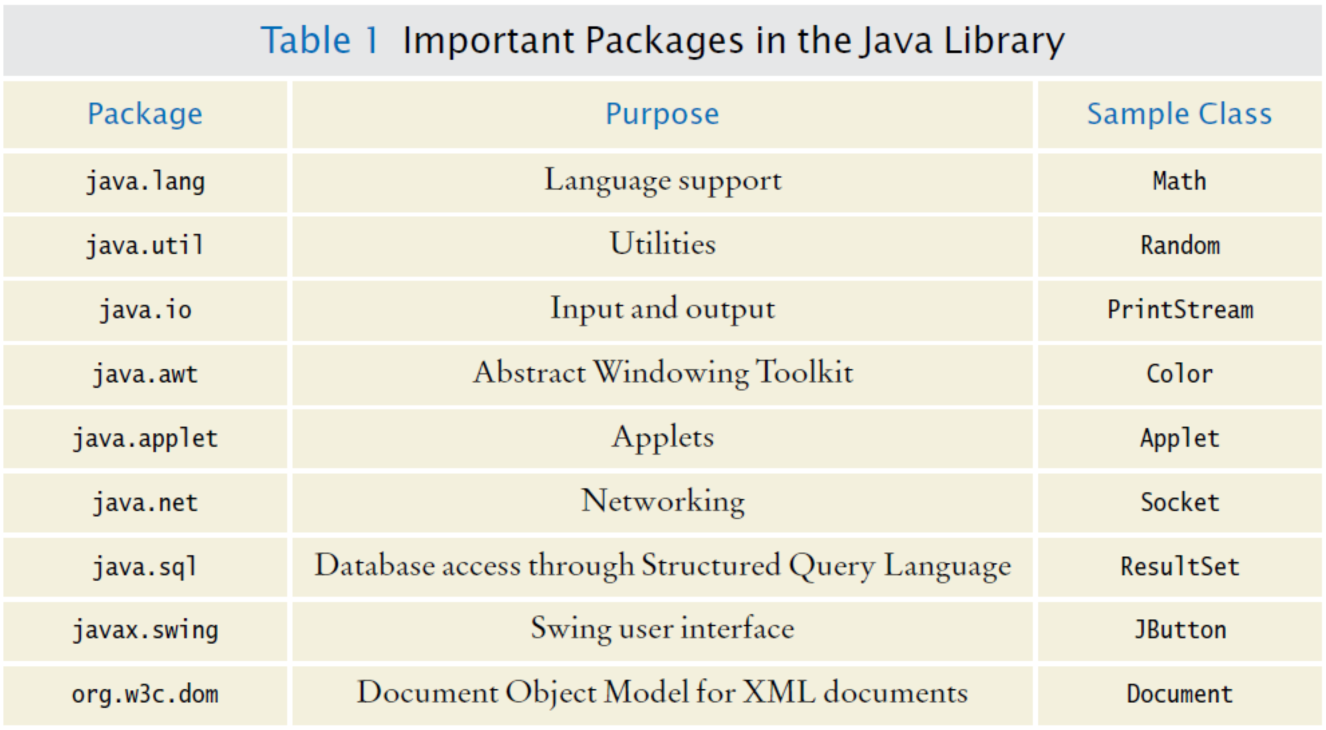


Table 1 Important Packages in the Java Library

Package	Purpose	Sample Class
`java.lang`	Language support	`Math`
`java.util`	Utilities	`Random`
`java.io`	Input and output	`PrintStream`
`java.awt`	Abstract Windowing Toolkit	`Color`
`java.applet`	Applets	`Applet`
`java.net`	Networking	`Socket`
`java.sql`	Database access through Structured Query Language	`ResultSet`
`javax.swing`	Swing user interface	`JButton`
`org.w3c.dom`	Document Object Model for XML documents	`Document`

To put one of your classes in a package, you must place a line

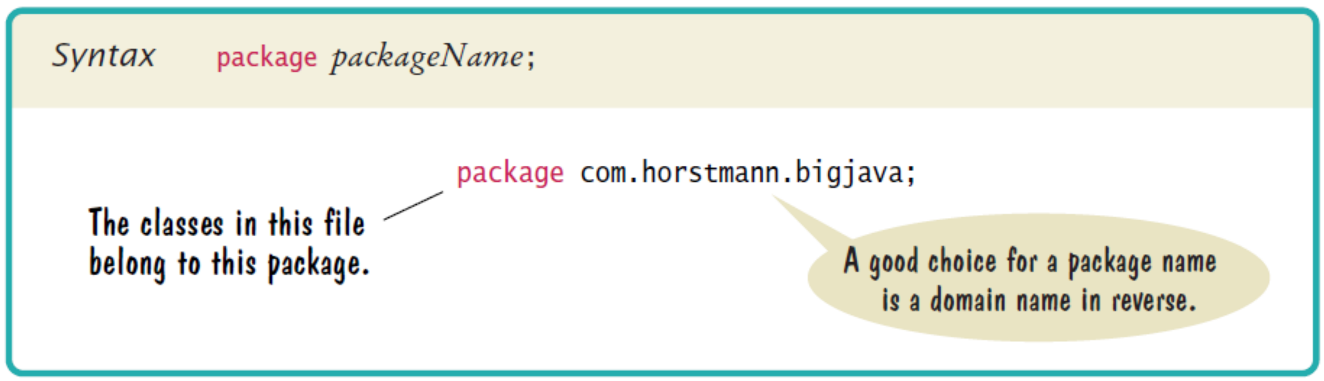
```
package packageName;
```

as the first instruction in the source file containing the class.

In addition to the named packages (such as java.util or com.horstmann.bigjava), there is a special package, called the default package, which has no name. If you did not include any package statement at the top of your source file, its classes are placed in the default package.

The import directive lets you refer to a class of a package by its class name, without the package prefix.

Package Specification

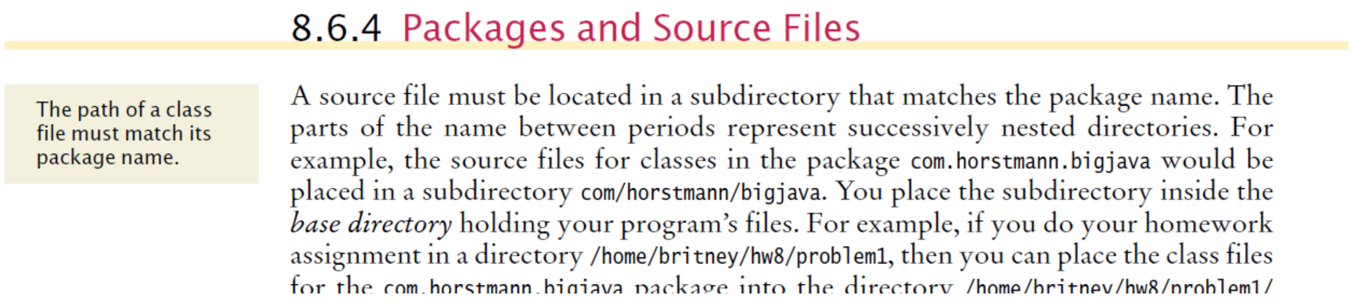


Syntax `package` *packageName*;

```
package com.horstmann.bigjava;
```

The classes in this file belong to this package.
A good choice for a package name is a domain name in reverse.

If you don't have your own domain name, you can still create a package name that has a high probability of being unique by writing your e-mail address backwards. For example, if Britney Walters has an e-mail address `walters@cs.sjsu.edu` , then she can use a package name edu.sjsu.cs.walters for her own classes.
Some instructors will want you to place each of your assignments into a separate package, such as homework1, homework2, and so on. The reason is again to avoid name collision. You can have two classes, homework1.Bank and homework2.Bank, with slightly different properties.



8.6.4 Packages and Source Files

The path of a class file must match its package name.

A source file must be located in a subdirectory that matches the package name. The parts of the name between periods represent successively nested directories. For example, the source files for classes in the package `com.horstmann.bigjava` would be placed in a subdirectory `com/horstmann/bigjava`. You place the subdirectory inside the *base directory* holding your program's files. For example, if you do your homework assignment in a directory `/home/britney/hw8/problem1`, then you can place the class files for the `com.horstmann.bigjava` package into the directory `/home/britney/hw8/problem1/`

`com/horstmann/bigjava`, as shown in Figure 6. (Here, we are using UNIX-style file names. Under Windows, you might use `c:\Users\Britney\hw8\problem1\com\horstmann\bigjava`.)

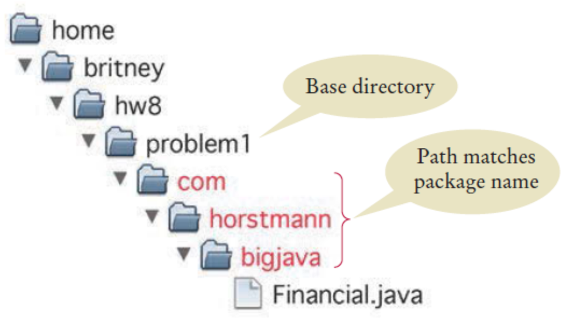


Figure 6 Base Directories and Subdirectories for Packages

An instance variable or method that is not declared as public or private can be accessed by all classes in the same package, which is usually not desirable.

Unit Test FrameWorks

Unit test frameworks simplify the task of writing classes that contain many test cases.

Unit testing frameworks were designed to quickly execute and evaluate test suites and to make it easy to incrementally add test cases. One of the most popular testing frameworks is JUnit. It is freely available at http://junit.org, and it is also built into a number of development environments, including BlueJ and Eclipse. Here we describe JUnit 4, the most current version of the library as this book is written. When you use JUnit, you design a companion test class for each class that you develop. You provide a method for each test case that you want to have executed. You use “annotations” to mark the test methods.
An annotation is an advanced Java feature that places a marker into the code that is interpreted by another tool. In the case of JUnit, the @Test annotation is used to mark test methods. In each test case, you make some computations and then compute some condition that you believe to be true. You then pass the result to a method that communicates a test result to the framework, **most commonly the assertEquals method. The assertEquals method takes as arguments the expected and actual values and, for floatingpoint numbers, a tolerance value.**

It is also customary (but not required) that the name of the test class ends in Test, such as

CashRegisterTest.
Here is a typical example:

```
import org.junit.Test;
import org.junit.Assert;
public class CashRegisterTest
{
    @Test public void twoPurchases()
    {
        CashRegister register = new CashRegister();
        register.recordPurchase(0.75);
        register.recordPurchase(1.50);
        register.receivePayment(2, 0, 5, 0, 0);
        double expected = 0.25;
        Assert.assertEquals(expected, register.giveChange(), EPSILON);
    }
    // More test cases
    . . .
}
```

The JUnit philosophy is to run all tests whenever you change your code.

Inheritance

Inheritance is a relationship between a more general class (called the superclass) and a more specialized class (called the subclass).
The subclass inherits data and behavior from the superclass.

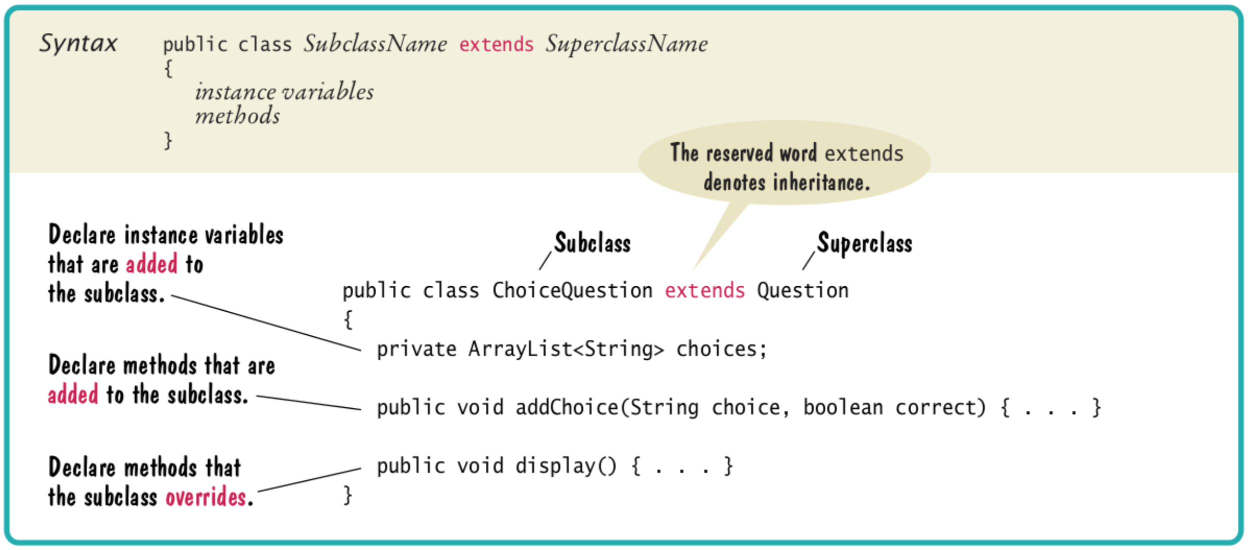
The **substitution principle** states that you can always use a subclass object when a superclass object is expected. A subclass inherits all methods that it does not override. Subclass objects automatically have the instance variables that are declared in the superclass. You only declare instance variables that are not part of the superclass objects.

The subclass inherits all Media public methods from the superclass.
You declare any methods that are new to the subclass, and change the implementation of inherited methods if the inherited behavior is not appropriate. When you supply a new implementation for an inherited method, you override the method.

```
public class ChoiceQuestion extends Question
{
    // This instance variable is added to the subclass
    private ArrayList<String> choices;

    // This method is added to the subclass
    public void addChoice(String choice, boolean correct) { . . . }

    // This method overrides a method from the superclass
    public void display() { . . . }
}
```

The reserved word extends denotes inheritance - indicates that a class inherits from a superclass.

Syntax 9.1 Subclass Declaration



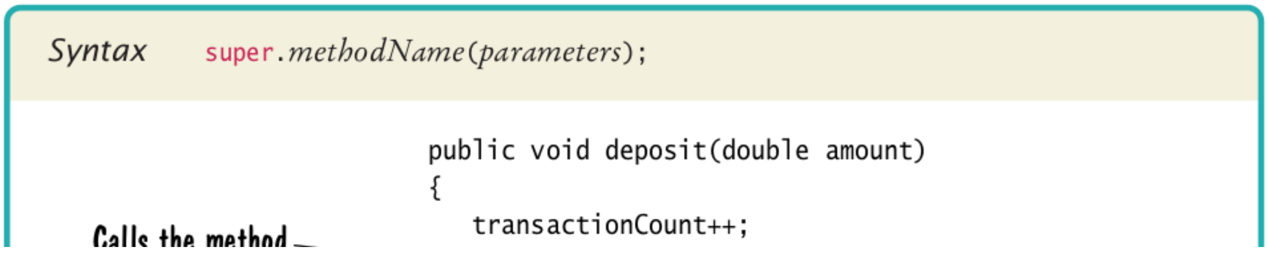
```
Syntax    public class SubclassName extends SuperclassName
          {
             instance variables
             methods
          }
```

```
public class ChoiceQuestion extends Question
{
   private ArrayList<String> choices;

   public void addChoice(String choice, boolean correct) { . . . }

   public void display() { . . . }
}
```

The reserved word extends denotes inheritance.
Subclass
Superclass
Declare instance variables that are added to the subclass.
Declare methods that are added to the subclass.
Declare methods that the subclass overrides.

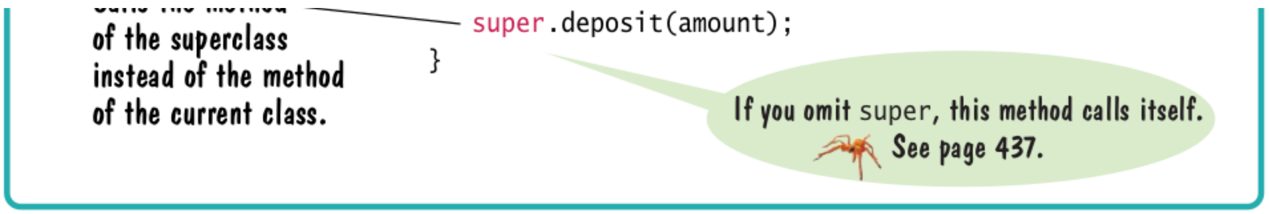
Override Methods from SuperClass

Calling a Superclass Method



```
Syntax    super.methodName(parameters);
```

```
public void deposit(double amount)
{
   transactionCount++;
```

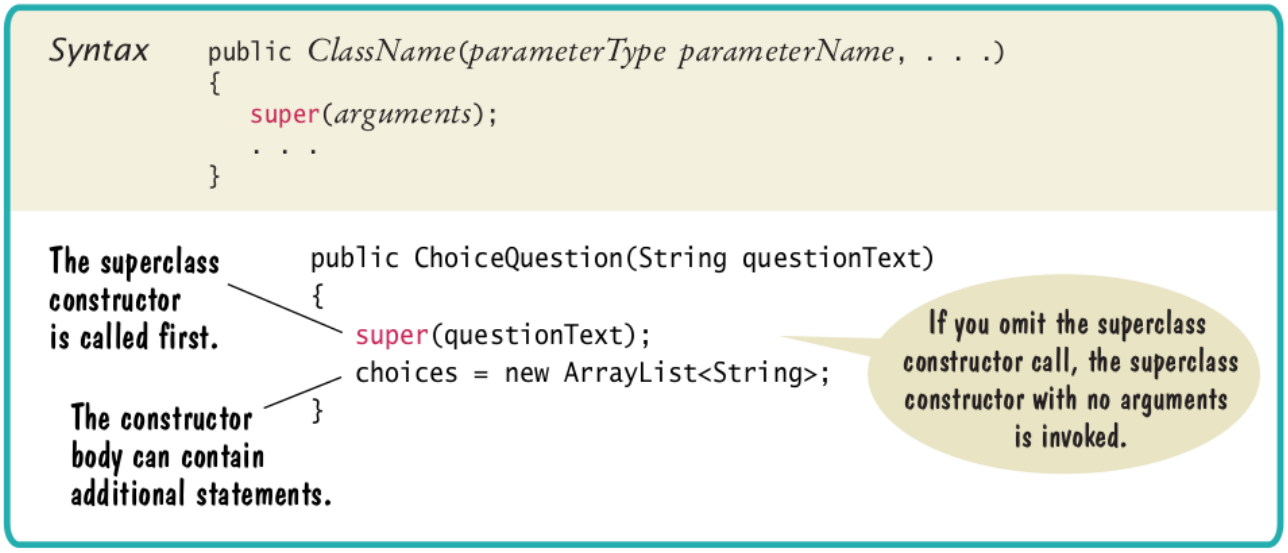


```
    super.deposit(amount);
}
```

of the superclass instead of the method of the current class.
If you omit super, this method calls itself. See page 437.

Unless specified otherwise, the subclass constructor calls the superclass constructor with no arguments. To call a superclass constructor, use the super reserved word in the first statement of the subclass constructor.

Constructor with Superclass Initializer



Syntax

```
public ClassName(parameterType parameterName, . . .)
{
   super(arguments);
   . . .
}
```

```
public ChoiceQuestion(String questionText)
{
   super(questionText);
   choices = new ArrayList<String>;
}
```

The superclass constructor is called first.
The constructor body can contain additional statements.
If you omit the superclass constructor call, the superclass constructor with no arguments is invoked.

Polymorphism

When you extend an existing class, you have the choice whether or not to override the methods of the superclass. Sometimes, it is desirable to force programmers to override a method.
That happens when there is no good default for the superclass and only the subclass programmer can know how to implement the method properly.

```
 public abstract void deductFees();
```

An abstract method has no implementation. This forces the implementors of subclasses to specify concrete implementations of this method. (Of course, some subclasses might decide to

implement a do-nothing method, but then that is their choice—not a silently inherited default.)
You cannot construct objects of classes with abstract methods. For example, once the Account class has an abstract method, the compiler will flag an attempt to create a new Account() as an error.
A class for which you cannot create objects is called an abstract class. A class for which you can create objects is sometimes called a concrete class. In Java, you must declare all abstract classes with the reserved word abstract :

```
public abstract class Account
    {
        public abstract void deductFees();
        . . .
        }
        public class SavingsAccount extends Account // Not abstract
        {
        . . .
        public void deductFees() // Provides an implementation
        {
        . . .
    }
}
```

If a class extends an abstract class without providing an implementation of all abstract methods, it too is abstract.

```
public abstract class BusinessAccount
{
// No implementation of deductFees
}
```

Note that you cannot construct an object of an abstract class, but you can still have an object reference whose type is an abstract class. Of course, the actual object to which it refers must be an instance of a concrete subclass:

```
Account anAccount; // OK
anAccount = new Account(); // Error— Account is
anAccount = new SavingsAccount(); // OK
anAccount = null; // OK
```

Special Topic 9.4

Final Methods and Classes

In Special Topic 9.3 you saw how you can force other programmers to create subclasses of abstract classes and override abstract methods. Occasionally, you may want to do the opposite and *prevent* other programmers from creating subclasses or from overriding certain methods. In these situations, you use the `final` reserved word. For example, the `String` class in the standard Java library has been declared as

```
public final class String { . . . }
```

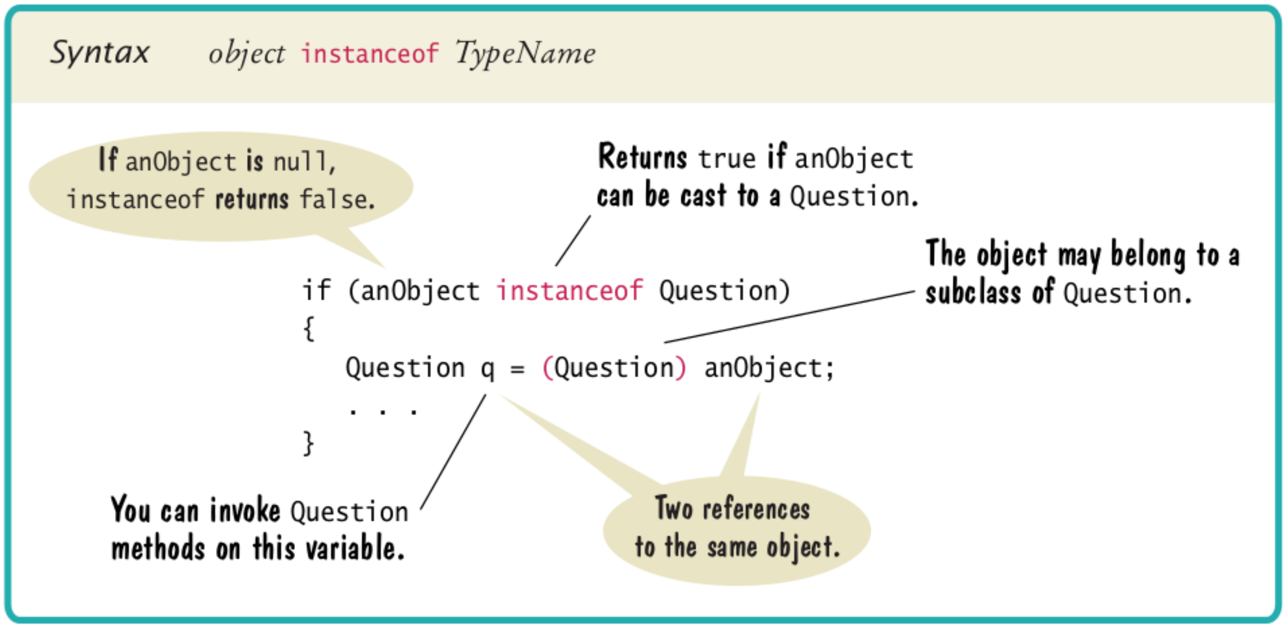
That means that nobody can extend the `String` class. When you have a reference of type `String`, it must contain a `String` object, never an object of a subclass.

You can also declare individual methods as final:

```
public class SecureAccount extends BankAccount
{
   . . .
   public final boolean checkPassword(String password)
   {
      . . .
   }
}
```

This way, nobody can override the `checkPassword` method with another method that simply returns `true`.

The instanceof Operator



Syntax *object* `instanceof` *TypeName*

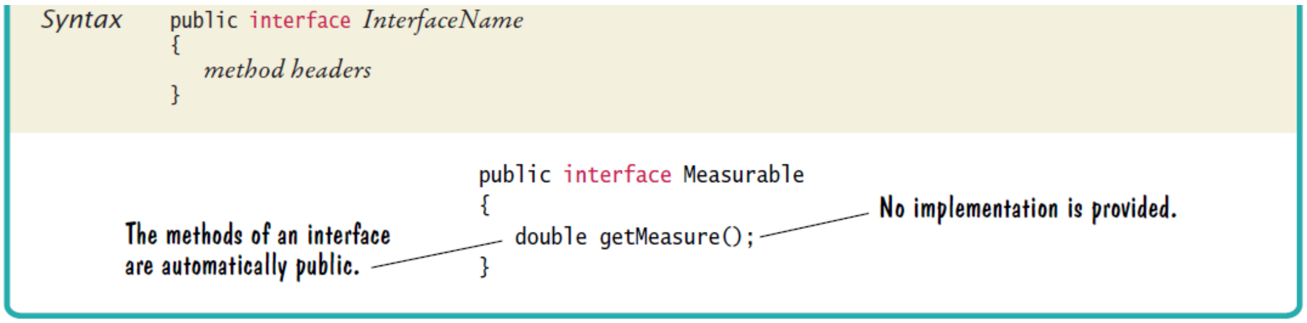
```
if (anObject instanceof Question)
{
   Question q = (Question) anObject;
   . . .
}
```

If anObject is null, instanceof returns false.
Returns true if anObject can be cast to a Question.
The object may belong to a subclass of Question.
You can invoke Question methods on this variable.
Two references to the same object.

Interfaces

A Java interface type declares the methods that can be applied to a variable of that type.

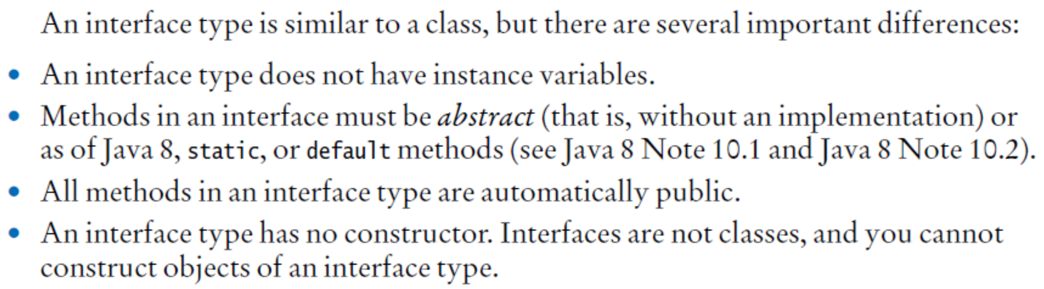
Syntax 10.1 Declaring an Interface



```
Syntax    public interface InterfaceName
          {
             method headers
          }
```

```
public interface Measurable
{
   double getMeasure();
}
```

The methods of an interface are automatically public.
No implementation is provided.



An interface type is similar to a class, but there are several important differences:

- An interface type does not have instance variables.
- Methods in an interface must be *abstract* (that is, without an implementation) or as of Java 8, `static`, or `default` methods (see Java 8 Note 10.1 and Java 8 Note 10.2).
- All methods in an interface type are automatically public.
- An interface type has no constructor. Interfaces are not classes, and you cannot construct objects of an interface type.

Now that we have a type that denotes measurability, we can implement a reusable *average method*:

```
public static double average(Measurable[] objects)
{
    double sum = 0;
    for (Measurable obj : objects)
    {
        sum = sum + obj.getMeasure();
    }
    if (objects.length > 0) { return sum / objects.length; }
    else { return 0; }
}
```

Use the **implements** reserved word to indicate that a class implements an interface type.

The **average** method of the preceding section can process objects of any class that implements the **Measurable** interface. A class **implements** an interface type if it declares the interface in an implements clause, like this:

```
public class BankAccount implements Measurable
```

The class should then implement the abstract method or methods that the interface requires:

```
public class BankAccount implements Measurable
{
    . . .
    public double getMeasure()
    {
    return balance;
    }
}
```

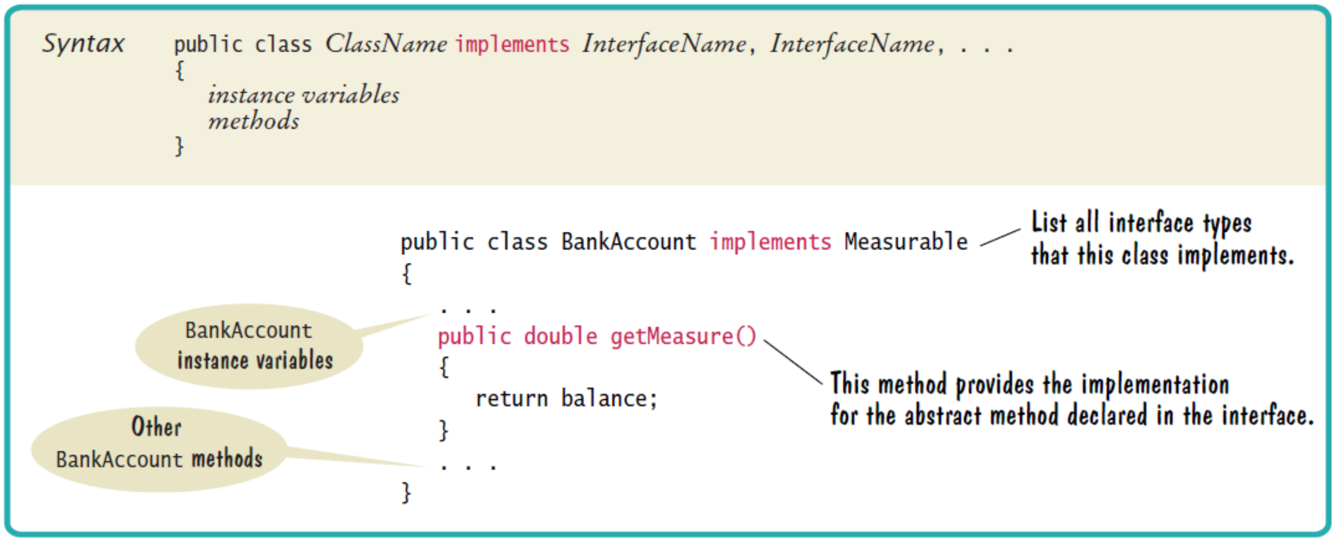
Note that the class must declare the method as **public**, whereas the interface need not—all methods in an interface are public.

Once the **BankAccount** class implements the **Measurable** interface type, **BankAccount** objects are instances of the **Measurable** type:

```
Measurable obj = new BankAccount(); // OK
```

A variable of type **Measurable** holds a reference to an object of some class that implements the **Measurable** interface.

Syntax 10.2 Implementing an Interface



Syntax

```
public class ClassName implements InterfaceName, InterfaceName, . . .
{
   instance variables
   methods
}
```

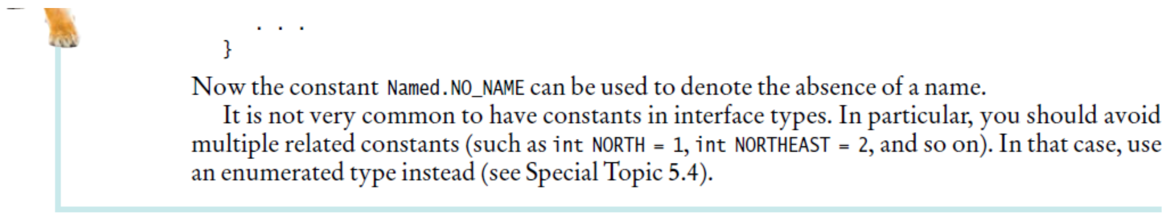
```
public class BankAccount implements Measurable
{
   . . .
   public double getMeasure()
   {
      return balance;
   }
   . . .
}
```

List all interface types that this class implements.
BankAccount instance variables
This method provides the implementation for the abstract method declared in the interface.
Other BankAccount methods

Special Topic 10.1 Constants in Interfaces

Interfaces cannot have instance variables, but it is legal to specify constants. When declaring a constant in an interface, you can (and should) omit the reserved words `public static final`, because all variables in an interface are automatically `public static final`. For example,

```
public interface Named
{
   String NO_NAME = "(NONE)";
```



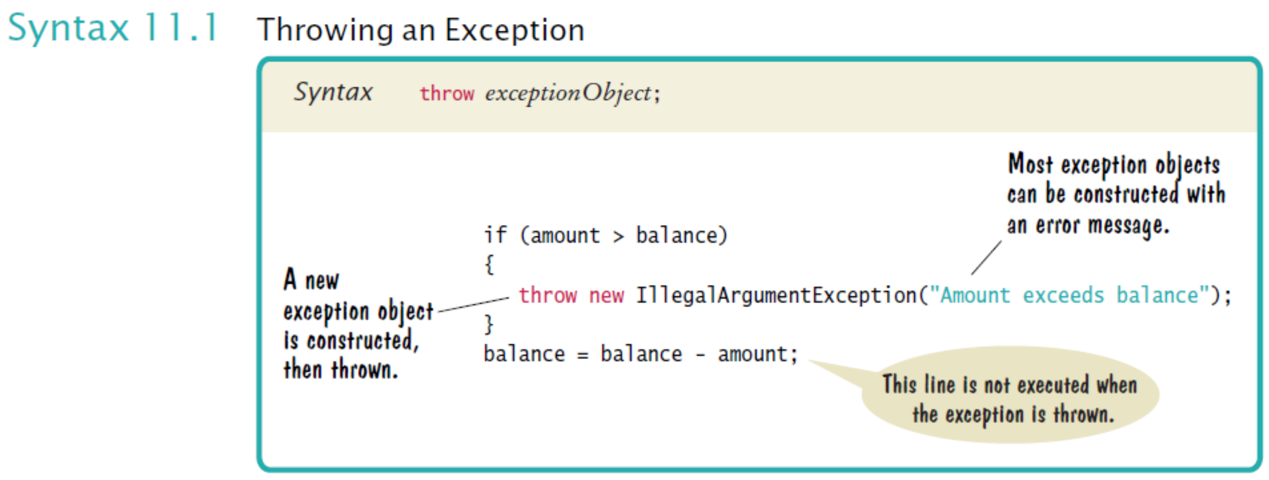
```
        . . .
    }
```

Now the constant Named.NO_NAME can be used to denote the absence of a name.
It is not very common to have constants in interface types. In particular, you should avoid multiple related constants (such as int NORTH = 1, int NORTHEAST = 2, and so on). In that case, use an enumerated type instead (see Special Topic 5.4).

```
See 10.3 for the **Comparable Interface** and **clone** method.
```

Throw Exceptions

To signal an exceptional condition, use the throw statement to throw an exception object.



Syntax 11.1 Throwing an Exception

Syntax `throw` *exceptionObject*;

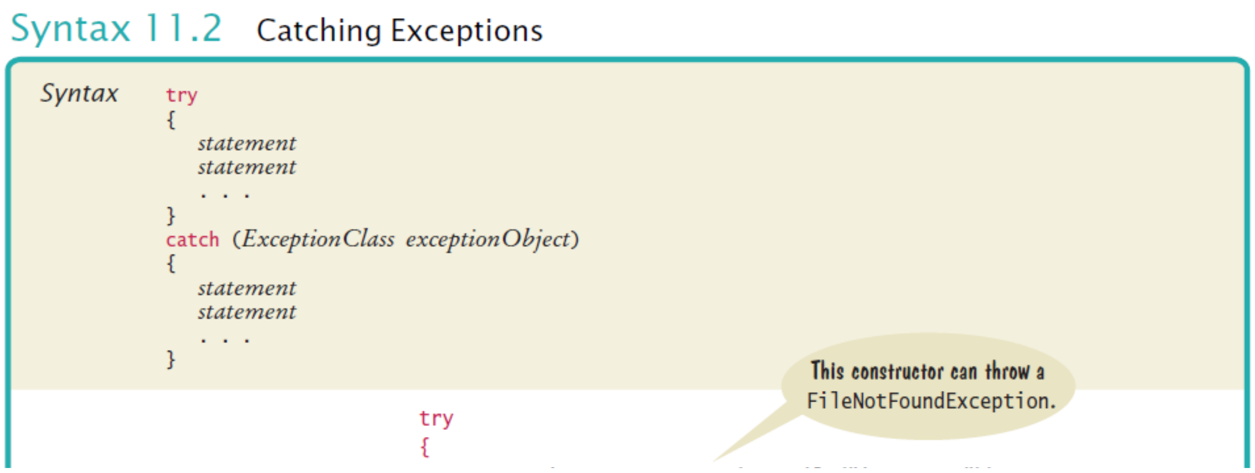
```
if (amount > balance)
{
   throw new IllegalArgumentException("Amount exceeds balance");
}
balance = balance - amount;
```

Most exception objects can be constructed with an error message.

A new exception object is constructed, then thrown.

This line is not executed when the exception is thrown.

Place the statements that can cause an exception inside a try block, and the handler inside a catch clause.

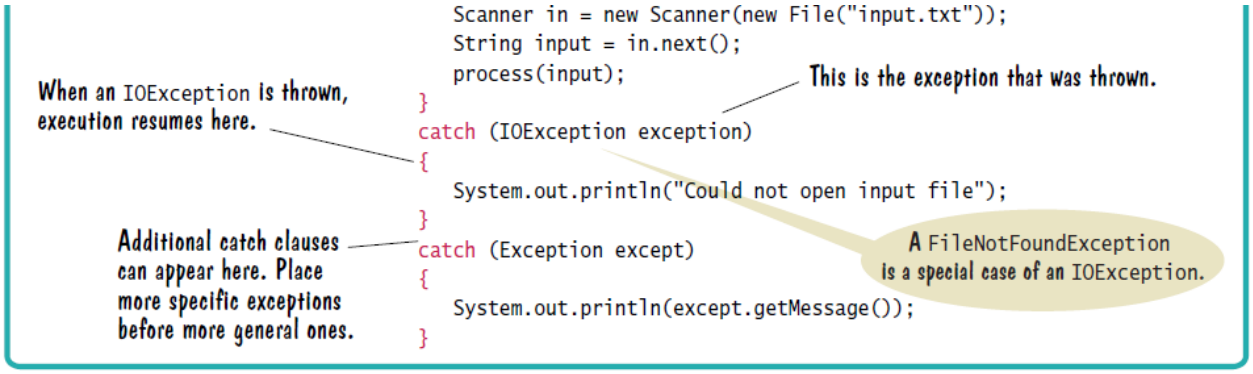


Syntax 11.2 Catching Exceptions

Syntax
```
try
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   . . .
}
```

```
try
{
```

This constructor can throw a FileNotFoundException.



```
    Scanner in = new Scanner(new File("input.txt"));
    String input = in.next();
    process(input);
}
catch (IOException exception)
{
    System.out.println("Could not open input file");
}
catch (Exception except)
{
    System.out.println(except.getMessage());
}
```

When an IOException is thrown, execution resumes here.

This is the exception that was thrown.

A FileNotFoundException is a special case of an IOException.

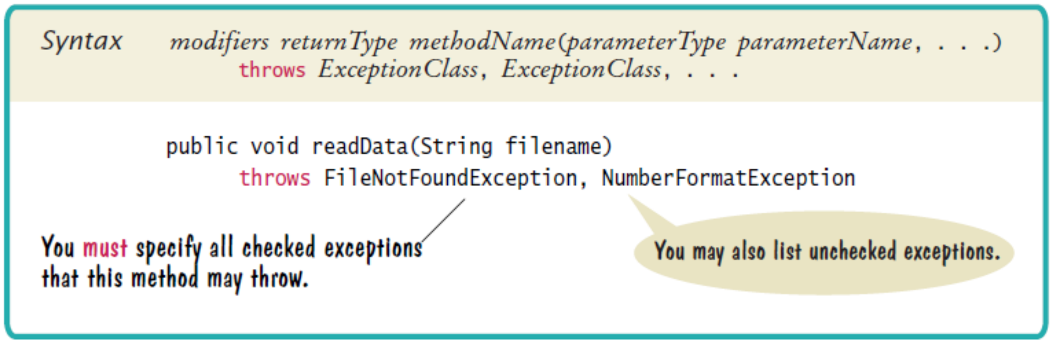
Additional catch clauses can appear here. Place more specific exceptions before more general ones.

Add a throws clause to a method that can throw a checked exception.

However, it commonly happens that the current method cannot handle the exception. In that case, you need to tell the compiler that you are aware of this exception and that you want your method to be terminated when it occurs. You supply the method with a throws clause:

```
public void readData(String filename) throws FileNotFoundException
{
    File inFile = new File(filename);
    Scanner in = new Scanner(inFile);
    . . .
}
```

Syntax 11.3 The throws Clause



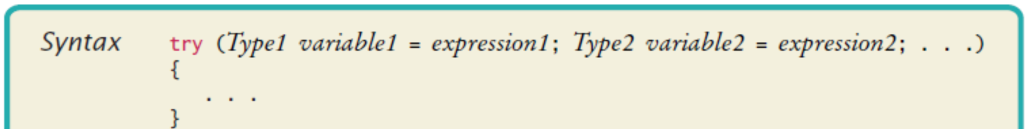
Syntax *modifiers returnType methodName(parameterType parameterName, . . .)*
    throws *ExceptionClass, ExceptionClass, . . .*

```
public void readData(String filename)
      throws FileNotFoundException, NumberFormatException
```

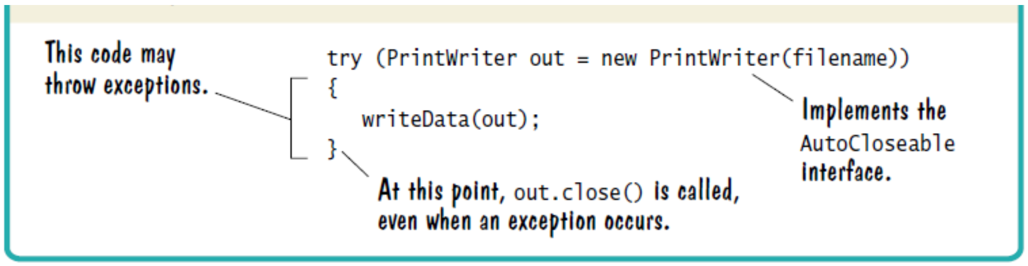
You must specify all checked exceptions that this method may throw.

You may also list unchecked exceptions.

Syntax 11.4 The try-with-resources Statement



Syntax try (*Type1 variable1 = expression1; Type2 variable2 = expression2; . . .*)
{
   . . .
}



```
try (PrintWriter out = new PrintWriter(filename))
{
   writeData(out);
}
```

This code may throw exceptions.
Implements the AutoCloseable interface.
At this point, out.close() is called, even when an exception occurs.

Special Topic 11.6

Assertions

An assertion is a condition that you believe to be true at all times in a particular program location. An assertion check tests whether an assertion is true. Here is a typical assertion check:

```
public double deposit (double amount)
{
   assert amount >= 0;
   balance = balance + amount;
}
```

In this method, the programmer expects that the quantity amount can never be negative. When the assertion is correct, no harm is done, and the program works in the normal way. If, for some reason, the assertion fails, and assertion checking is enabled, then the `assert` statement throws an exception of type `AssertionError`, causing the program to terminate.

However, if assertion checking is disabled, then the assertion is never checked, and the program runs at full speed. By default, assertion checking is disabled when you execute a program.

To execute a program with assertion checking turned on, use this command:

```
java -enableassertions MainClass
```

You can also use the shortcut `-ea` instead of `-enableassertions`. You should turn assertion checking on during program development and testing.

Special Topic 11.7

The `try/finally` Statement

You saw in Section 11.4 how to ensure that a resource is closed when an exception occurs. The `try`-with-resources statement calls the `close` methods of variables declared within the statement header. You should always use the `try`-with-resources statement when closing resources.

It can happen that you need to do some cleanup other than calling a `close` method. In that case, use the `try/finally` statement:

```
public double deposit (double amount)
try
{
   . . .
}
finally
{
   Cleanup. // This code is executed whether or not an exception occurs
}
```

If the body of the `try` statement completes without an exception, the cleanup happens. If an exception is thrown, the cleanup happens and the exception is then propagated to its handler.

The `try/finally` statement is rarely required because most Java library classes that require cleanup implement the `AutoCloseable` interface. However, you will see a use of this statement in Chapter 22.