Java Tutorials

Updated for Java SE 8

The Java Tutorials

Learning the Java Language

Table of Contents

Object-Oriented Programming Concepts

What Is an Object?

What Is a Class?

What Is Inheritance?

What Is an Interface?

What Is a Package?

Questions and Exercises

Answers to Questions and Exercises

Language Basics

Variables

Primitive Data Types

Arrays

Java Language Keywords

Summary of Variables

Questions and Exercises

Answers to Questions and Exercises

Operators

Assignment

Equality

Bitwise and Bit Shift Operators

Summary of Operators

Questions and Exercises

Answers to Questions and Exercises

Expressions

Questions and Exercises

Answers to Questions and Exercises

Control Flow Statements

The if-then and if-then-else Statements

The switch Statement

The while and do-while Statements

The for Statement

Branching Statements

Summary of Control Flow Statements

Questions and Exercises

Answers to Questions and Exercises

Classes and Objects

Classes

Declaring Classes

Declaring Member Variables

Defining Methods

Providing Constructors for Your Classes

Passing Information to a Method or a Constructor

Objects

Creating Objects

Using Objects

More on Classes

Returning a Value from a Method

Using the this Keyword

Controlling Access to Members of a Class

Understanding Class Members

Initializing Fields

Summary of Creating and Using Classes and Objects

Questions and Exercises

Answers to Questions and Exercises

Questions and Exercises

Answers to Questions and Exercises

Nested Classes

Inner Class Example

Local Classes

Anonymous Classes

Lambda Expressions

Method References

When to Use Nested Classes

Questions and Exercises

Answers to Questions and Exercises

Enum Types

Questions and Exercises

Answers to Questions and Exercises

Annotations

Annotations Basics

Declaring an Annotation Type

Predefined Annotation Types

Type Annotations and Pluggable Type Systems

Repeating Annotations

Questions and Exercises

Answers to Questions and Exercises

Interfaces and Inheritance

Interfaces

Defining an Interface

Implementing an Interface

Using an Interface as a Type

Evolving Interfaces

Default Methods

Summary of Interfaces

Questions and Exercises

Answers to Questions and Exercises

Inheritance

Multiple Inheritance of State

Overriding and Hiding Methods

Polymorphism

Hiding Fields

Using the Keyword super

Object as a Superclass

Writing Final Classes and Methods

Abstract Methods and Classes

Summary of Inheritance

Questions and Exercises

Answers to Questions and Exercises

Numbers and Strings

Numbers

The Numbers Classes

Formatting Numeric Print Output

Beyond Basic Arithmetic

Summary of Numbers

Questions and Exercises

Answers to Questions and Exercises

Characters

Strings

Converting Between Numbers and Strings

Manipulating Characters in a String

Comparing Strings and Portions of Strings

The StringBuilder Class

Summary of Characters and Strings

Autoboxing and Unboxing

Questions and Exercises

Answers to Questions and Exercises

Generics (Updated)

Why Use Generics?

Generic Types

Raw Types

Generic Methods

Bounded Type Parameters

Generic Methods and Bounded Type Parameters

Generics

Type Inference

Wildcards

Upper Bounded Wildcards

Unbounded Wildcards

Lower Bounded Wildcards

Wildcards and Subtyping

Wildcard Capture and Helper Methods

Guidelines for Wildcard Use

Type Erasure

Erasure of Generic Types

Erasure of Generic Methods

Effects of Type Erasure and Bridge Methods

Non-Reifiable Types

Restrictions on Generics

Questions and Exercises

Answer to Questions and Exercises

Packages

Creating and Using Packages

Creating a Package

Naming a Package

Using Package Members

Managing Source and Class Files

Summary of Creating and Using Packages

Questions and Exercises

Answers to Questions and Exercises

End of Trail

Java Tutorials

The Java Tutorials are a one stop shop for learning the Java language. The tutorials are practical guides for programmers who want to use the Java programming language to create applets and applications. They also serve as a reference for the experienced Java programmer.

Groups of related lessons are organized into "trails"; for example, the "Getting Started" trail, or the "Learning the Java Language" trail. This ebook contains a single trail. You can download other trails via the link on the Java Tutorials home page.

To read an ebook file that you have downloaded, transfer the document to your device.

We would love to hear what you think of this ebook, including any problems that you find. Please send us your feedback.

Legal Notices

Supported Platforms

The `mobi` file format works best on the following devices:

- Kindle Fire
- Kindle DX

The `ePub` file format works best on the following devices:

- iPad
- Nook
- Other eReaders that support the `ePub` format.

For best results when viewing preformatted code blocks, adjust the landscape/portrait orientation and font size of your device to enable the maximum possible viewing area.

Trail: Learning the Java Language

This trail covers the fundamentals of programming in the Java programming language.

Object-Oriented Programming Concepts teaches you the core concepts behind object-oriented programming: objects, messages, classes, and inheritance. This lesson ends by showing you how these concepts translate into code. Feel free to skip this lesson if you are already familiar with object-oriented programming.

Language Basics describes the traditional features of the language, including variables, arrays, data types, operators, and control flow.

Classes and Objects describes how to write the classes from which objects are created, and how to create and use the objects.

Annotations are a form of metadata and provide information for the compiler. This lesson describes where and how to use annotations in a program effectively.

Interfaces and Inheritance describes interfaces—what they are, why you would want to write one, and how to write one. This section also describes the way in which you can derive one class from another. That is, how a *subclass* can inherit fields and methods from a *superclass*. You will learn that all classes are derived from the `Object` class, and how to modify the methods that a subclass inherits from superclasses.

Numbers and Strings This lesson describes how to use `Number` and `String` objects The lesson also shows you how to format data for output.

Generics are a powerful feature of the Java programming language. They improve the type safety of your code, making more of your bugs detectable at compile time.

Packages are a feature of the Java programming language that help you to organize and structure your classes and their relationships to one another.

Trail: Learning the Java Language: Table of Contents

- Object-Oriented Programming Concepts
  - What Is an Object?
  - What Is a Class?
  - What Is Inheritance?
  - What Is an Interface?
  - What Is a Package?
  - Questions and Exercises: Object-Oriented Programming Concepts
- Language Basics
  - Variables
    - Primitive Data Types
    - Arrays
    - Summary of Variables
  - Questions and Exercises: Variables
  - Operators
    - Assignment, Arithmetic, and Unary Operators
    - Equality, Relational, and Conditional Operators
    - Bitwise and Bit Shift Operators
    - Summary of Operators
  - Questions and Exercises: Operators
  - Expressions, Statements, and Blocks
  - Questions and Exercises: Expressions, Statements, and Blocks
  - Control Flow Statements
    - The if-then and if-then-else Statements
    - The switch Statement
    - The while and do-while Statements
    - The for Statement
    - Branching Statements
    - Summary of Control Flow Statements
  - Questions and Exercises: Control Flow Statements
- Classes and Objects
  - Classes
    - Declaring Classes
    - Declaring Member Variables
    - Defining Methods
    - Providing Constructors for Your Classes
    - Passing Information to a Method or a Constructor
  - Objects
    - Creating Objects
    - Using Objects
  - More on Classes
    - Returning a Value from a Method
    - Using the this Keyword
    - Controlling Access to Members of a Class

            - Understanding Class Members
            - Initializing Fields
            - Summary of Creating and Using Classes and Objects
        - Questions and Exercises: Classes
        - Questions and Exercises: Objects
        - Nested Classes
            - Inner Class Example
            - Local Classes
            - Anonymous Classes
            - Lambda Expressions
                - Method References
            - When to Use Nested Classes, Local Classes, Anonymous Classes, and Lambda Expressions
        - Questions and Exercises: Nested Classes
        - Enum Types
        - Questions and Exercises: Enum Types
- Annotations
    - Annotations Basics
    - Declaring an Annotation Type
    - Predefined Annotation Types
    - Type Annotations and Pluggable Type Systems
    - Repeating Annotations
    - Questions and Exercises: Annotations
- Interfaces and Inheritance
    - Interfaces
        - Defining an Interface
        - Implementing an Interface
        - Using an Interface as a Type
        - Evolving Interfaces
        - Default Methods
        - Summary of Interfaces
    - Questions and Exercises: Interfaces
    - Inheritance
        - Multiple Inheritance of State, Implementation, and Type
        - Overriding and Hiding Methods
        - Polymorphism
        - Hiding Fields
        - Using the Keyword super
        - Object as a Superclass
        - Writing Final Classes and Methods
        - Abstract Methods and Classes
        - Summary of Inheritance
    - Questions and Exercises: Inheritance
- Numbers and Strings
    - Numbers

        - The Numbers Classes
        - Formatting Numeric Print Output
        - Beyond Basic Arithmetic
        - Summary of Numbers
    - Questions and Exercises: Numbers
    - Characters
    - Strings
        - Converting Between Numbers and Strings
        - Manipulating Characters in a String
        - Comparing Strings and Portions of Strings
        - The StringBuilder Class
        - Summary of Characters and Strings
    - Autoboxing and Unboxing
    - Questions and Exercises: Characters and Strings
- Generics (Updated)
    - Why Use Generics?
    - Generic Types
        - Raw Types
    - Generic Methods
    - Bounded Type Parameters
        - Generic Methods and Bounded Type Parameters
    - Generics, Inheritance, and Subtypes
    - Type Inference
    - Wildcards
        - Upper Bounded Wildcards
        - Unbounded Wildcards
        - Lower Bounded Wildcards
        - Wildcards and Subtyping
        - Wildcard Capture and Helper Methods
        - Guidelines for Wildcard Use
    - Type Erasure
        - Erasure of Generic Types
        - Erasure of Generic Methods
        - Effects of Type Erasure and Bridge Methods
        - Non-Reifiable Types
    - Restrictions on Generics
    - Questions and Exercises: Generics
- Packages
    - Creating and Using Packages
        - Creating a Package
        - Naming a Package
        - Using Package Members
        - Managing Source and Class Files
        - Summary of Creating and Using Packages
    - Questions and Exercises: Creating and Using Packages

Lesson: Object-Oriented Programming Concepts

If you've never used an object-oriented programming language before, you'll need to learn a few basic concepts before you can begin writing any code. This lesson will introduce you to objects, classes, inheritance, interfaces, and packages. Each discussion focuses on how these concepts relate to the real world, while simultaneously providing an introduction to the syntax of the Java programming language.

What Is an Object?

An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. This lesson explains how state and behavior are represented within an object, introduces the concept of data encapsulation, and explains the benefits of designing your software in this manner.

What Is a Class?

A class is a blueprint or prototype from which objects are created. This section defines a class that models the state and behavior of a real-world object. It intentionally focuses on the basics, showing how even a simple class can cleanly model state and behavior.

What Is Inheritance?

Inheritance provides a powerful and natural mechanism for organizing and structuring your software. This section explains how classes inherit state and behavior from their superclasses, and explains how to derive one class from another using the simple syntax provided by the Java programming language.

What Is an Interface?

An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface. This section defines a simple interface and explains the necessary changes for any class that implements it.

What Is a Package?

A package is a namespace for organizing classes and interfaces in a logical manner. Placing your code into packages makes large software projects easier to manage. This section explains why this is useful, and introduces you to the Application Programming Interface (API) provided by the Java platform.

Questions and Exercises: Object-Oriented Programming Concepts

Use the questions and exercises presented in this section to test your understanding of objects, classes, inheritance, interfaces, and packages.

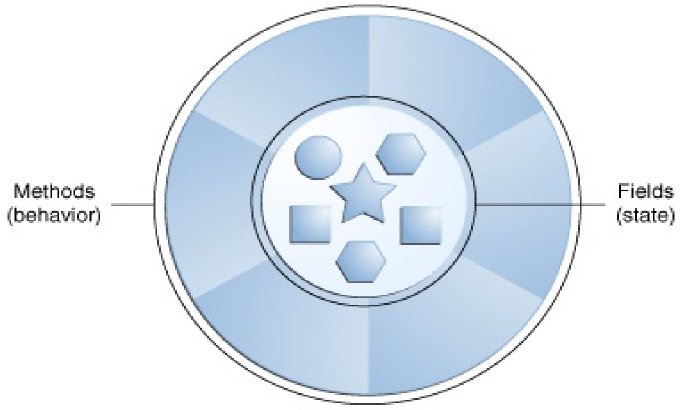
Note: See online version of topics in this ebook to download complete source code.

What Is an Object?

Objects are key to understanding *object-oriented* technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

Real-world objects share two characteristics: They all have *state* and *behavior*. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

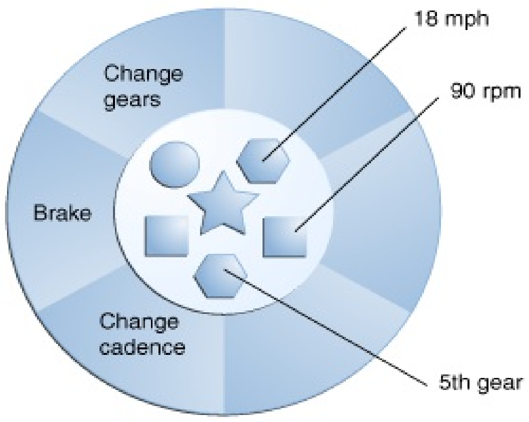
Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions: "What possible states can this object be in?" and "What possible behavior can this object perform?". Make sure to write down your observations. As you do, you'll notice that real-world objects vary in complexity; your desktop lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). You may also notice that some objects, in turn, will also contain other objects. These real-world observations all translate into the world of object-oriented programming.



A software object.

Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in *fields* (variables in some programming languages) and exposes its behavior through *methods* (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. Hiding internal state and requiring all interaction to be performed through an object's methods is known as *data encapsulation* — a fundamental principle of object-oriented programming.

Consider a bicycle, for example:



A bicycle modeled as a software object.

By attributing state (current speed, current pedal cadence, and current gear) and providing methods for changing that state, the object remains in control of how the outside world is allowed to use it. For example, if the bicycle only has 6 gears, a method to change gears could reject any value that is less than 1 or greater than 6.

Bundling code into individual software objects provides a number of benefits, including:

1. Modularity: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
2. Information-hiding: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
3. Code re-use: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
4. Pluggability and debugging ease: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace *it*, not the entire machine.

What Is a Class?

In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an *instance* of the *class of objects* known as bicycles. A *class* is the blueprint from which individual objects are created.

The following `Bicycle` class is one possible implementation of a bicycle:

```java
class Bicycle {

    int cadence = 0;
    int speed = 0;
    int gear = 1;

    void changeCadence(int newValue) {
         cadence = newValue;
    }

    void changeGear(int newValue) {
         gear = newValue;
    }

    void speedUp(int increment) {
         speed = speed + increment;
    }

    void applyBrakes(int decrement) {
         speed = speed - decrement;
    }

    void printStates() {
         System.out.println("cadence:" +
             cadence + " speed:" +
             speed + " gear:" + gear);
    }
}
```

The syntax of the Java programming language will look new to you, but the design of this class is based on the previous discussion of bicycle objects. The fields `cadence`, `speed`, and `gear` represent the object's state, and the methods (`changeCadence`, `changeGear`, `speedUp` etc.) define its interaction with the outside world.

You may have noticed that the `Bicycle` class does not contain a `main` method. That's because it's not a complete application; it's just the blueprint for bicycles that might be *used* in an application. The responsibility of creating and using new `Bicycle` objects belongs to some other class in your application.

Here's a `BicycleDemo` class that creates two separate `Bicycle` objects and invokes their methods:

```java
class BicycleDemo {
    public static void main(String[] args) {
        // Create two different
        // Bicycle objects
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();

        // Invoke methods on
        // those objects
        bike1.changeCadence(50);
        bike1.speedUp(10);
        bike1.changeGear(2);
        bike1.printStates();

        bike2.changeCadence(50);
        bike2.speedUp(10);
        bike2.changeGear(2);
        bike2.changeCadence(40);
        bike2.speedUp(10);
        bike2.changeGear(3);
        bike2.printStates();
    }
}
```

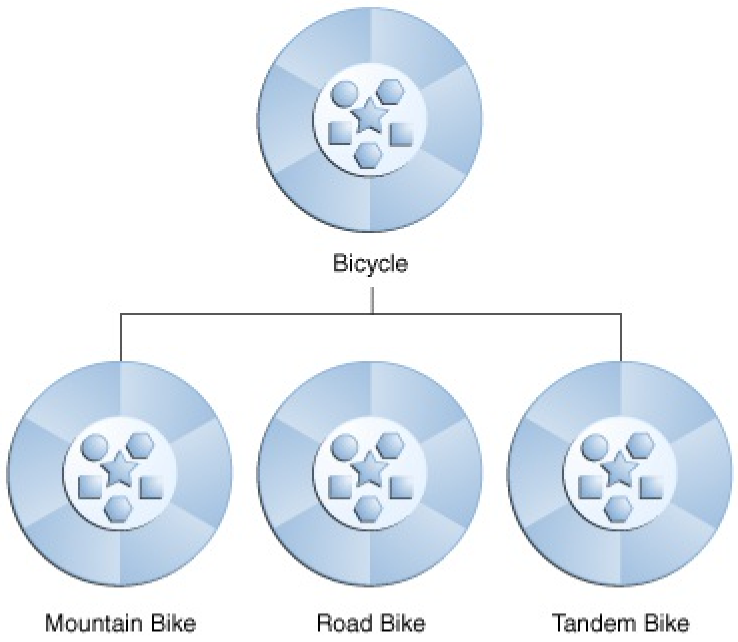
The output of this test prints the ending pedal cadence, speed, and gear for the two bicycles:

```
cadence:50 speed:10 gear:2
cadence:40 speed:20 gear:3
```

What Is Inheritance?

Different kinds of objects often have a certain amount in common with each other. Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear). Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio.

Object-oriented programming allows classes to *inherit* commonly used state and behavior from other classes. In this example, `Bicycle` now becomes the *superclass* of `MountainBike`, `RoadBike`, and `TandemBike`. In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of *subclasses*:



A hierarchy of bicycle classes.

The syntax for creating a subclass is simple. At the beginning of your class declaration, use the `extends` keyword, followed by the name of the class to inherit from:

```
class MountainBike extends Bicycle {

    // new fields and methods defining
    // a mountain bike would go here

}
```

This gives `MountainBike` all the same fields and methods as `Bicycle`, yet allows its code to focus exclusively on the features that make it unique. This makes code for your subclasses easy to read. However, you must take care to properly document the state and behavior that each superclass defines, since that code will not appear in the source file of each subclass.

What Is an Interface?

As you've already learned, objects define their interaction with the outside world through the methods that they expose. Methods form the object's *interface* with the outside world; the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off.

In its most common form, an interface is a group of related methods with empty bodies. A bicycle's behavior, if specified as an interface, might appear as follows:

```
interface Bicycle {

    //  wheel revolutions per minute
    void changeCadence(int newValue);

    void changeGear(int newValue);

    void speedUp(int increment);

    void applyBrakes(int decrement);
}
```

To implement this interface, the name of your class would change (to a particular brand of bicycle, for example, such as `ACMEBicycle`), and you'd use the `implements` keyword in the class declaration:

```
class ACMEBicycle implements Bicycle {

    int cadence = 0;
    int speed = 0;
    int gear = 1;

   // The compiler will now require that methods
   // changeCadence, changeGear, speedUp, and applyBrakes
   // all be implemented. Compilation will fail if those
   // methods are missing from this class.

    void changeCadence(int newValue) {
         cadence = newValue;
    }

    void changeGear(int newValue) {
         gear = newValue;
    }

    void speedUp(int increment) {
         speed = speed + increment;
    }

    void applyBrakes(int decrement) {
         speed = speed - decrement;
    }

    void printStates() {
         System.out.println("cadence:" +
             cadence + " speed:" +
             speed + " gear:" + gear);
    }
}
```

Implementing an interface allows a class to become more formal about the behavior it promises to provide. Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler. If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile.

Note: To actually compile the `ACMEBicycle` class, you'll need to add the `public` keyword to the beginning of the implemented interface methods. You'll learn the reasons for this later in the lessons on Classes and Objects and Interfaces and Inheritance.

What Is a Package?

A package is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another. Because software written in the Java programming language can be composed of hundreds or *thousands* of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.

The Java platform provides an enormous class library (a set of packages) suitable for use in your own applications. This library is known as the "Application Programming Interface", or "API" for short. Its packages represent the tasks most commonly associated with general-purpose programming. For example, a `String` object contains state and behavior for character strings; a `File` object allows a programmer to easily create, delete, inspect, compare, or modify a file on the filesystem; a `Socket` object allows for the creation and use of network sockets; various GUI objects control buttons and checkboxes and anything else related to graphical user interfaces. There are literally thousands of classes to choose from. This allows you, the programmer, to focus on the design of your particular application, rather than the infrastructure required to make it work.

The Java Platform API Specification contains the complete listing for all packages, interfaces, classes, fields, and methods supplied by the Java SE platform. Load the page in your browser and bookmark it. As a programmer, it will become your single most important piece of reference documentation.

Questions and Exercises: Object-Oriented Programming Concepts

Questions

1. Real-world objects contain ___ and ___.
2. A software object's state is stored in ___.
3. A software object's behavior is exposed through ___.
4. Hiding internal data from the outside world, and accessing it only through publicly exposed methods is known as data ___.
5. A blueprint for a software object is called a ___.
6. Common behavior can be defined in a ___ and inherited into a ___ using the ___ keyword.
7. A collection of methods with no implementation is called an ___.
8. A namespace that organizes classes and interfaces by functionality is called a ___.
9. The term API stands for ___?

Exercises

1. Create new classes for each real-world object that you observed at the beginning of this trail. Refer to the Bicycle class if you forget the required syntax.
2. For each new class that you've created above, create an interface that defines its behavior, then require your class to implement it. Omit one or two methods and try compiling. What does the error look like?

Check your answers.

Answers to Questions

1. Real-world objects contain **state** and **behavior**.
2. A software object's state is stored in **fields**.
3. A software object's behavior is exposed through **methods**.
4. Hiding internal data from the outside world, and accessing it only through publicly exposed methods is known as data **encapsulation**.
5. A blueprint for a software object is called a **class**.
6. Common behavior can be defined in a **superclass** and inherited into a **subclass** using the **extends** keyword.
7. A collection of methods with no implementation is called an **interface**.
8. A namespace that organizes classes and interfaces by functionality is called a **package**.
9. The term API stands for **Application Programming Interface**.

Answers to Exercises

1. Your answers will vary depending on the real-world objects that you are modeling.
2. Your answers will vary here as well, but the error message will specifically list the required methods that have not been implemented.

Lesson: Language Basics

Variables

You've already learned that objects store their state in fields. However, the Java programming language also uses the term "variable" as well. This section discusses this relationship, plus variable naming rules and conventions, basic data types (primitive types, character strings, and arrays), default values, and literals.

Operators

This section describes the operators of the Java programming language. It presents the most commonly-used operators first, and the less commonly-used operators last. Each discussion includes code samples that you can compile and run.

Expressions, Statements, and Blocks

Operators may be used in building expressions, which compute values; expressions are the core components of statements; statements may be grouped into blocks. This section discusses expressions, statements, and blocks using example code that you've already seen.

Control Flow Statements

This section describes the control flow statements supported by the Java programming language. It covers the decisions-making, looping, and branching statements that enable your programs to conditionally execute particular blocks of code.

Note: See online version of topics in this ebook to download complete source code.

Variables

As you learned in the previous lesson, an object stores its state in *fields*.

```
int cadence = 0;
int speed = 0;
int gear = 1;
```

The What Is an Object? discussion introduced you to fields, but you probably have still a few questions, such as: What are the rules and conventions for naming a field? Besides `int`, what other data types are there? Do fields have to be initialized when they are declared? Are fields assigned a default value if they are not explicitly initialized? We'll explore the answers to such questions in this lesson, but before we do, there are a few technical distinctions you must first become aware of. In the Java programming language, the terms "field" and "variable" are both used; this is a common source of confusion among new developers, since both often seem to refer to the same thing.

The Java programming language defines the following kinds of variables:

- **Instance Variables (Non-Static Fields)** Technically speaking, objects store their individual states in "non-static fields", that is, fields declared without the `static` keyword. Non-static fields are also known as *instance variables* because their values are unique to each *instance* of a class (to each object, in other words); the `currentSpeed` of one bicycle is independent from the `currentSpeed` of another.
- **Class Variables (Static Fields)** A *class variable* is any field declared with the `static` modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated. A field defining the number of gears for a particular kind of bicycle could be marked as `static` since conceptually the same number of gears will apply to all instances. The code `static int numGears = 6;` would create such a static field. Additionally, the keyword `final` could be added to indicate that the number of gears will never change.
- **Local Variables** Similar to how an object stores its state in fields, a method will often store its temporary state in *local variables*. The syntax for declaring a local variable is similar to declaring a field (for example, `int count = 0;`). There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared — which is between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.
- **Parameters** You've already seen examples of parameters, both in the `Bicycle` class and in the `main` method of the "Hello World!" application. Recall that the signature for the `main` method is `public static void main(String[] args)`. Here, the `args` variable is the parameter to this method. The important thing to remember is that parameters are always classified as "variables" not "fields". This applies to other parameter-accepting constructs as well (such as constructors and exception handlers) that you'll learn about later in the tutorial.

Having said that, the remainder of this tutorial uses the following general guidelines when discussing fields and variables. If we are talking about "fields in general" (excluding local variables and parameters), we may simply say "fields". If the discussion applies to "all of the above", we may simply say "variables". If the context calls for a distinction, we will use specific terms (static field, local variables, etc.) as appropriate. You may also occasionally see the term "member" used as well. A type's fields, methods, and nested types are collectively called its *members*.

Naming

Every programming language has its own set of rules and conventions for the kinds of names that you're allowed to use, and the Java programming language is no different. The rules and conventions for naming your variables can be summarized as follows:

- Variable names are case-sensitive. A variable's name can be any legal identifier — an unlimited-length sequence of Unicode letters and digits, beginning with a letter, the dollar sign "`$`", or the underscore character "`_`". The convention, however, is to always begin your variable names with a letter, not "`$`" or "`_`". Additionally, the dollar sign character, by convention, is never used at all. You may find some situations where auto-generated names will contain the dollar sign, but your variable names should always avoid using it. A similar convention exists for the underscore character; while it's technically legal to begin your variable's name with "`_`", this practice is discouraged. White space is not permitted.
- Subsequent characters may be letters, digits, dollar signs, or underscore characters. Conventions (and common sense) apply to this rule as well. When choosing a name for your variables, use full words instead of cryptic abbreviations. Doing so will make your code easier to read and understand. In many cases it will also make your code self-documenting; fields named `cadence`, `speed`, and `gear`, for example, are much more intuitive than abbreviated versions, such as `s`, `c`, and `g`. Also keep in mind that the name you choose must not be a keyword or reserved word.
- If the name you choose consists of only one word, spell that word in all lowercase letters. If it consists of more than one word, capitalize the first letter of each subsequent word. The names `gearRatio` and `currentGear` are prime examples of this convention. If your variable stores a constant value, such as `static final int NUM_GEARS = 6`, the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character. By convention, the underscore character is never used elsewhere.

Primitive Data Types

The Java programming language is statically-typed, which means that all variables must first be declared before they can be used. This involves stating the variable's type and name, as you've already seen:

```
int gear = 1;
```

Doing so tells your program that a field named "gear" exists, holds numerical data, and has an initial value of "1". A variable's data type determines the values it may contain, plus the operations that may be performed on it. In addition to `int`, the Java programming language supports seven other *primitive data types*. A primitive type is predefined by the language and is named by a reserved keyword. Primitive values do not share state with other primitive values. The eight primitive data types supported by the Java programming language are:

- **byte**: The `byte` data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The `byte` data type can be useful for saving memory in large arrays, where the memory savings actually matters. They can also be used in place of `int` where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.
- **short**: The `short` data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with `byte`, the same guidelines apply: you can use a `short` to save memory in large arrays, in situations where the memory savings actually matters.
- **int**: By default, the `int` data type is a 32-bit signed two's complement integer, which has a minimum value of -2^{31} and a maximum value of 2^{31}-1. In Java SE 8 and later, you can use the `int` data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of 2^{32}-1. Use the Integer class to use `int` data type as an unsigned integer. See the section The Number Classes for more information. Static methods like `compareUnsigned`, `divideUnsigned` etc have been added to the `Integer` class to support the arithmetic operations for unsigned integers.
- **long**: The `long` data type is a 64-bit two's complement integer. The signed long has a minimum value of -2^{63} and a maximum value of 2^{63}-1. In Java SE 8 and later, you can use the `long` data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of 2^{64}-1. The unsigned long has a minimum value of 0 and maximum value of 2^{64}-1. Use this data type when you need a range of values wider than those provided by `int`. The `Long` class also contains methods like `compareUnsigned`, `divideUnsigned` etc to support arithmetic operations for unsigned long.
- **float**: The `float` data type is a single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification. As with the recommendations for `byte` and `short`, use a `float` (instead of `double`) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency. For that, you will need to use the java.math.BigDecimal class instead. Numbers and

Strings covers `BigDecimal` and other useful classes provided by the Java platform.
- **double**: The `double` data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification. For decimal values, this data type is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.
- **boolean**: The `boolean` data type has only two possible values: `true` and `false`. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.
- **char**: The `char` data type is a single 16-bit Unicode character. It has a minimum value of `'\u0000'` (or 0) and a maximum value of `'\uffff'` (or 65,535 inclusive).

In addition to the eight primitive data types listed above, the Java programming language also provides special support for character strings via the java.lang.String class. Enclosing your character string within double quotes will automatically create a new `String` object; for example, `String s = "this is a string";`. `String` objects are *immutable*, which means that once created, their values cannot be changed. The `String` class is not technically a primitive data type, but considering the special support given to it by the language, you'll probably tend to think of it as such. You'll learn more about the `String` class in Simple Data Objects

Default Values

It's not always necessary to assign a value when a field is declared. Fields that are declared but not initialized will be set to a reasonable default by the compiler. Generally speaking, this default will be zero or `null`, depending on the data type. Relying on such default values, however, is generally considered bad programming style.

The following chart summarizes the default values for the above data types.

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Local variables are slightly different; the compiler never assigns a default value to an uninitialized local variable. If you cannot initialize your local variable where it is declared, make sure to assign it a value before you attempt to use it. Accessing an uninitialized local variable will result in a compile-time error.

Literals

You may have noticed that the `new` keyword isn't used when initializing a variable of a primitive type. Primitive types are special data types built into the language; they are not objects created from a class. A *literal* is the source code representation of a fixed value; literals are represented directly in your code without requiring computation. As shown below, it's possible to assign a literal to a variable of a primitive type:

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

Integer Literals

An integer literal is of type `long` if it ends with the letter `L` or `l`; otherwise it is of type `int`. It is recommended that you use the upper case letter `L` because the lower case letter `l` is hard to distinguish from the digit `1`.

Values of the integral types `byte`, `short`, `int`, and `long` can be created from `int` literals. Values of type `long` that exceed the range of `int` can be created from `long` literals. Integer literals can be expressed by these number systems:

- Decimal: Base 10, whose digits consists of the numbers 0 through 9; this is the number system you use every day
- Hexadecimal: Base 16, whose digits consist of the numbers 0 through 9 and the letters A through F
- Binary: Base 2, whose digits consists of the numbers 0 and 1 (you can create binary literals in Java SE 7 and later)

For general-purpose programming, the decimal system is likely to be the only number system you'll ever use. However, if you need to use another number system, the following example shows the correct syntax. The prefix `0x` indicates hexadecimal and `0b` indicates binary:

```
// The number 26, in decimal
int decVal = 26;
//  The number 26, in hexadecimal
int hexVal = 0x1a;
// The number 26, in binary
int binVal = 0b11010;
```

Floating-Point Literals

A floating-point literal is of type `float` if it ends with the letter `F` or `f`; otherwise its type is `double` and it can optionally end with the letter `D` or `d`.

The floating point types (`float` and `double`) can also be expressed using E or e (for scientific notation), F or f (32-bit float literal) and D or d (64-bit double literal; this is the default and by convention is omitted).

```
double d1 = 123.4;
// same value as d1, but in scientific notation
double d2 = 1.234e2;
float f1  = 123.4f;
```

Character and String Literals

Literals of types `char` and `String` may contain any Unicode (UTF-16) characters. If your editor and file system allow it, you can use such characters directly in your code. If not, you can use a "Unicode escape" such as `'\u0108'` (capital C with circumflex), or `"S\u00ED Se\u00F1or"` (Sí Seor in Spanish). Always use 'single quotes' for `char` literals and "double quotes" for `String` literals. Unicode escape sequences may be used elsewhere in a program (such as in field names, for example), not just in `char` or `String` literals.

The Java programming language also supports a few special escape sequences for `char` and `String` literals: `\b` (backspace), `\t` (tab), `\n` (line feed), `\f` (form feed), `\r` (carriage return), `\"` (double quote), `\'` (single quote), and `\\` (backslash).

There's also a special `null` literal that can be used as a value for any reference type. `null` may be assigned to any variable, except variables of primitive types. There's little you can do with a `null` value beyond testing for its presence. Therefore, `null` is often used in programs as a marker to indicate that some object is unavailable.

Finally, there's also a special kind of literal called a *class literal*, formed by taking a type name and appending "`.class`"; for example, `String.class`. This refers to the object (of type `Class`) that represents the type itself.

Using Underscore Characters in Numeric Literals

In Java SE 7 and later, any number of underscore characters (_) can appear anywhere between digits in a numerical literal. This feature enables you, for example. to separate groups of digits in numeric literals, which can improve the readability of your code.

For instance, if your code contains numbers with many digits, you can use an underscore character to separate digits in groups of three, similar to how you would use a punctuation mark like a comma, or a space, as a separator.

The following example shows other ways you can use the underscore in numeric literals:

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi =  3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

You can place underscores only between digits; you cannot place underscores in the following places:

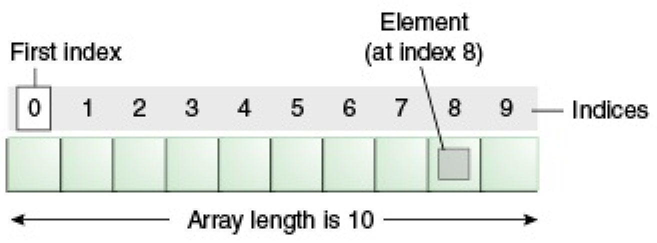
- At the beginning or end of a number
- Adjacent to a decimal point in a floating point literal
- Prior to an `F` or `L` suffix
- In positions where a string of digits is expected

The following examples demonstrate valid and invalid underscore placements (which are highlighted) in numeric literals:

```
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi1 = 3_.1415F;
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi2 = 3._1415F;
// Invalid: cannot put underscores
// prior to an L suffix
long socialSecurityNumber1 = 999_99_9999_L;

// OK (decimal literal)
int x1 = 5_2;
// Invalid: cannot put underscores
// At the end of a literal
int x2 = 52_;
// OK (decimal literal)
int x3 = 5_______2;

// Invalid: cannot put underscores
// in the 0x radix prefix
int x4 = 0_x52;
// Invalid: cannot put underscores
// at the beginning of a number
int x5 = 0x_52;
// OK (hexadecimal literal)
int x6 = 0x5_2;
// Invalid: cannot put underscores
// at the end of a number
int x7 = 0x52_;
```

Arrays

An *array* is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed. You have seen an example of arrays already, in the `main` method of the "Hello World!" application. This section discusses arrays in greater detail.



An array of 10 elements.

Each item in an array is called an *element*, and each element is accessed by its numerical *index*. As shown in the preceding illustration, numbering begins with 0. The 9th element, for example, would therefore be accessed at index 8.

The following program, `ArrayDemo`, creates an array of integers, puts some values in the array, and prints each value to standard output.

```
class ArrayDemo {
    public static void main(String[] args) {
        // declares an array of integers
        int[] anArray;

        // allocates memory for 10 integers
        anArray = new int[10];

        // initialize first element
        anArray[0] = 100;
        // initialize second element
        anArray[1] = 200;
        // and so forth
        anArray[2] = 300;
        anArray[3] = 400;
        anArray[4] = 500;
        anArray[5] = 600;
        anArray[6] = 700;
        anArray[7] = 800;
        anArray[8] = 900;
        anArray[9] = 1000;

        System.out.println("Element at index 0: "
                           + anArray[0]);
        System.out.println("Element at index 1: "
                           + anArray[1]);
        System.out.println("Element at index 2: "
                           + anArray[2]);
        System.out.println("Element at index 3: "
                           + anArray[3]);
        System.out.println("Element at index 4: "
                           + anArray[4]);
        System.out.println("Element at index 5: "
                           + anArray[5]);
        System.out.println("Element at index 6: "
                           + anArray[6]);
        System.out.println("Element at index 7: "
                           + anArray[7]);
        System.out.println("Element at index 8: "
                           + anArray[8]);
        System.out.println("Element at index 9: "
                           + anArray[9]);
    }
}
```

The output from this program is:

```
Element at index 0: 100
Element at index 1: 200
Element at index 2: 300
Element at index 3: 400
Element at index 4: 500
Element at index 5: 600
Element at index 6: 700
Element at index 7: 800
Element at index 8: 900
Element at index 9: 1000
```

In a real-world programming situation, you would probably use one of the supported *looping constructs* to iterate through each element of the array, rather than write each line individually as in the preceding example. However, the example clearly illustrates the array syntax. You will learn about the various looping constructs (`for`, `while`, and `do-while`) in the Control Flow section.

Declaring a Variable to Refer to an Array

The preceding program declares an array (named `anArray`) with the following line of code:

```
// declares an array of integers
int[] anArray;
```

Like declarations for variables of other types, an array declaration has two components: the array's type and the array's name. An array's type is written as *`type`*`[]`, where *`type`* is the data type of the contained elements; the brackets are special symbols indicating that this variable holds an array. The size of the array is not part of its type (which is why the brackets are empty). An array's name can be anything you want, provided that it follows the rules and conventions as previously discussed in the naming section. As with variables of other types, the declaration does not actually create an array; it simply tells the compiler that this variable will hold an array of the specified type.

Similarly, you can declare arrays of other types:

```
byte[] anArrayOfBytes;
short[] anArrayOfShorts;
long[] anArrayOfLongs;
float[] anArrayOfFloats;
double[] anArrayOfDoubles;
```

```
boolean[] anArrayOfBooleans;
char[] anArrayOfChars;
String[] anArrayOfStrings;
```

You can also place the brackets after the array's name:

```
// this form is discouraged
float anArrayOfFloats[];
```

However, convention discourages this form; the brackets identify the array type and should appear with the type designation.

Creating, Initializing, and Accessing an Array

One way to create an array is with the `new` operator. The next statement in the `ArrayDemo` program allocates an array with enough memory for 10 integer elements and assigns the array to the `anArray` variable.

```
// create an array of integers
anArray = new int[10];
```

If this statement is missing, then the compiler prints an error like the following, and compilation fails:

```
ArrayDemo.java:4: Variable anArray may not have been initialized.
```

The next few lines assign values to each element of the array:

```
anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
anArray[2] = 300; // and so forth
```

Each array element is accessed by its numerical index:

```
System.out.println("Element 1 at index 0: " + anArray[0]);
System.out.println("Element 2 at index 1: " + anArray[1]);
System.out.println("Element 3 at index 2: " + anArray[2]);
```

Alternatively, you can use the shortcut syntax to create and initialize an array:

```
int[] anArray = {
    100, 200, 300,
    400, 500, 600,
    700, 800, 900, 1000
};
```

Here the length of the array is determined by the number of values provided between braces and separated by commas.

You can also declare an array of arrays (also known as a *multidimensional* array) by using two or more sets of brackets, such as `String[][] names`. Each element, therefore, must be accessed by a corresponding number of index values.

In the Java programming language, a multidimensional array is an array whose components are themselves arrays. This is unlike arrays in C or Fortran. A consequence of this is that the rows are allowed to vary in length, as shown in the following `MultiDimArrayDemo` program:

```
class MultiDimArrayDemo {
    public static void main(String[] args) {
        String[][] names = {
            {"Mr. ", "Mrs. ", "Ms. "},
            {"Smith", "Jones"}
        };
        // Mr. Smith
        System.out.println(names[0][0] + names[1][0]);
        // Ms. Jones
        System.out.println(names[0][2] + names[1][1]);
    }
}
```

The output from this program is:

```
Mr. Smith
Ms. Jones
```

Finally, you can use the built-in `length` property to determine the size of any array. The following code prints the array's size to standard output:

```
 System.out.println(anArray.length);
```

Copying Arrays

The `System` class has an `arraycopy()` method that you can use to efficiently copy data from one array into another:

```
public static void arraycopy(Object src, int srcPos,
                             Object dest, int destPos, int length)
```

The two `Object` arguments specify the array to copy *from* and the array to copy *to*. The three `int` arguments specify the starting position in the source array, the starting position in the destination array, and the number of array elements to copy.

The following program, `ArrayCopyDemo`, declares an array of `char` elements, spelling the word "decaffeinated." It uses the `System.arraycopy()` method to copy a subsequence of array components into a second array:

```
class ArrayCopyDemo {
    public static void main(String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                            'i', 'n', 'a', 't', 'e', 'd' };
        char[] copyTo = new char[7];

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));
    }
}
```

The output from this program is:

```
caffein
```

Array Manipulations

Arrays are a powerful and useful concept used in programming. Java SE provides methods to perform some of the most common manipulations related to arrays. For instance, the `ArrayCopyDemo` example uses the `arraycopy()` method of the `System` class instead of manually iterating through the elements of the source array and placing each one into the destination array. This is performed behind the scenes, enabling the developer to use just one line of code to call the method.

For your convenience, Java SE provides several methods for performing array manipulations (common tasks, such as copying, sorting and searching arrays) in the `java.util.Arrays` class. For instance, the previous example can be modified to use the `CopyOfRange()` method of the `java.util.Arrays` class, as you can see in the `ArrayCopyOfDemo` example. The difference is that using the `CopyOfRange()` method does not require you to create the destination array before calling the method, because the destination array is returned by the method:

```
class ArrayCopyOfDemo {
    public static void main(String[] args) {

        char[] copyFrom = {'d', 'e', 'c', 'a', 'f', 'f', 'e',
            'i', 'n', 'a', 't', 'e', 'd'};

        char[] copyTo = java.util.Arrays.copyOfRange(copyFrom, 2, 9);

        System.out.println(new String(copyTo));
    }
}
```

As you can see, the output from this program is the same (`caffein`), although it requires fewer lines of code.

Some other useful operations provided by methods in the `java.util.Arrays` class, are:

- Searching an array for a specific value to get the index at which it is placed (the `binarySearch()` method).

- Comparing two arrays to determine if they are equal or not (the `equals()` method).
- Filling an array to place a specific value at each index (the `fill()` method).
- Sorting an array into ascending order. This can be done either sequentially, using the `sort()` method, or concurrently, using the `parallelSort()` method introduced in Java SE 8. Parallel sorting of large arrays on multiprocessor systems is faster than sequential array sorting.

Here is a list of keywords in the Java programming language. You cannot use any of the following as identifiers in your programs. The keywords `const` and `goto` are reserved, even though they are not currently used. `true`, `false`, and `null` might seem like keywords, but they are actually literals; you cannot use them as identifiers in your programs.

`abstract`	`continue`	`for`	`new`	`switch`
`assert`***	`default`	`goto`*	`package`	`synchronized`
`boolean`	`do`	`if`	`private`	`this`
`break`	`double`	`implements`	`protected`	`throw`
`byte`	`else`	`import`	`public`	`throws`
`case`	`enum`****	`instanceof`	`return`	`transient`
`catch`	`extends`	`int`	`short`	`try`
`char`	`final`	`interface`	`static`	`void`
`class`	`finally`	`long`	`strictfp`**	`volatile`
`const`*	`float`	`native`	`super`	`while`

*	not used
**	added in 1.2
***	added in 1.4
****	added in 5.0

Summary of Variables

The Java programming language uses both "fields" and "variables" as part of its terminology. Instance variables (non-static fields) are unique to each instance of a class. Class variables (static fields) are fields declared with the `static` modifier; there is exactly one copy of a class variable, regardless of how many times the class has been instantiated. Local variables store temporary state inside a method. Parameters are variables that provide extra information to a method; both local variables and parameters are always classified as "variables" (not "fields"). When naming your fields or variables, there are rules and conventions that you should (or must) follow.

The eight primitive data types are: byte, short, int, long, float, double, boolean, and char. The `java.lang.String` class represents character strings. The compiler will assign a reasonable default value for fields of the above types; for local variables, a default value is never assigned. A literal is the source code representation of a fixed value. An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed.

Questions and Exercises: Variables

Questions

1. The term "instance variable" is another name for ___.
2. The term "class variable" is another name for ___.
3. A local variable stores temporary state; it is declared inside a ___.
4. A variable declared within the opening and closing parenthesis of a method signature is called a ____.
5. What are the eight primitive data types supported by the Java programming language?
6. Character strings are represented by the class ___.
7. An ___ is a container object that holds a fixed number of values of a single type.

Exercises

1. Create a small program that defines some fields. Try creating some illegal field names and see what kind of error the compiler produces. Use the naming rules and conventions as a guide.
2. In the program you created in Exercise 1, try leaving the fields uninitialized and print out their values. Try the same with a local variable and see what kind of compiler errors you can produce. Becoming familiar with common compiler errors will make it easier to recognize bugs in your code.

Check your answers

Answers to Questions

1. The term "instance variable" is another name for **non-static field**.
2. The term "class variable" is another name for **static field**.
3. A local variable stores temporary state; it is declared inside a **method**.
4. A variable declared within the opening and closing parenthesis of a method is called a **parameter**.
5. What are the eight primitive data types supported by the Java programming language? **byte, short, int, long, float, double, boolean, char**
6. Character strings are represented by the class **java.lang.String**.
7. An **array** is a container object that holds a fixed number of values of a single type.

Answers to Exercises

1. Create a small program that defines some fields. Try creating some illegal field names and see what kind of error the compiler produces. Use the naming rules and conventions as a guide. There is no single correct answer here. Your results will vary depending on your code.
2. In the program you created in Exercise 1, try leaving the fields uninitialized and print out their values. Try the same with a local variable and see what kind of compiler errors you can produce. Becoming familiar with common compiler errors will make it easier to recognize bugs in your code. Again, there is no single correct answer for this exercise. Your results will vary depending on your code.

Operators

Now that you've learned how to declare and initialize variables, you probably want to know how to *do something* with them. Learning the operators of the Java programming language is a good place to start. Operators are special symbols that perform specific operations on one, two, or three *operands*, and then return a result.

As we explore the operators of the Java programming language, it may be helpful for you to know ahead of time which operators have the highest precedence. The operators in the following table are listed according to precedence order. The closer to the top of the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with relatively lower precedence. Operators on the same line have equal precedence. When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated from left to right; assignment operators are evaluated right to left.

Operator Precedence

Operators	**Precedence**
postfix	`expr++ expr--`
unary	`++expr --expr +expr -expr ~ !`
multiplicative	`* / %`
additive	`+ -`
shift	`<< >> >>>`
relational	`< > <= >= instanceof`
equality	`== !=`
bitwise AND	`&`
bitwise exclusive OR	`^`
bitwise inclusive OR	`|`
logical AND	`&&`
logical OR	`||`
ternary	`? :`
assignment	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

In general-purpose programming, certain operators tend to appear more frequently than others; for example, the assignment operator "=" is far more common than the unsigned right shift operator ">>>". With that in mind, the following discussion focuses first on the operators that you're most likely to use on a regular basis, and ends focusing on those that are less common. Each discussion is accompanied by sample code that you can compile and run. Studying its output will help reinforce what you've just learned.

Assignment, Arithmetic, and Unary Operators

The Simple Assignment Operator

One of the most common operators that you'll encounter is the simple assignment operator "=". You saw this operator in the Bicycle class; it assigns the value on its right to the operand on its left:

```
int cadence = 0;
int speed = 0;
int gear = 1;
```

This operator can also be used on objects to assign *object references*, as discussed in Creating Objects.

The Arithmetic Operators

The Java programming language provides operators that perform addition, subtraction, multiplication, and division. There's a good chance you'll recognize them by their counterparts in basic mathematics. The only symbol that might look new to you is "%", which divides one operand by another and returns the remainder as its result.

```
+       additive operator (also used for
        String concatenation)
-       subtraction operator
*       multiplication operator
/       division operator
%       remainder operator
```

The following program, `ArithmeticDemo`, tests the arithmetic operators.

```


class ArithmeticDemo {

    public static void main (String[] args){

        // result is now 3
        int result = 1 + 2;
        System.out.println(result);

        // result is now 2
        result = result - 1;
        System.out.println(result);

        // result is now 4
        result = result * 2;
        System.out.println(result);

        // result is now 2
        result = result / 2;
        System.out.println(result);

        // result is now 10
        result = result + 8;
        // result is now 3
        result = result % 7;
        System.out.println(result);
    }
}
```

You can also combine the arithmetic operators with the simple assignment operator to create *compound assignments*. For example, `x+=1;` and `x=x+1;` both increment the value of `x` by 1.

The `+` operator can also be used for concatenating (joining) two strings together, as shown in the following `ConcatDemo` program:

```
class ConcatDemo {
    public static void main(String[] args){
        String firstString = "This is";
        String secondString = " a concatenated string.";
        String thirdString = firstString+secondString;
        System.out.println(thirdString);
    }
}
```

By the end of this program, the variable `thirdString` contains "This is a concatenated string.", which gets printed to standard output.

The Unary Operators

The unary operators require only one operand; they perform various operations such as incrementing/decrementing a value by one, negating an expression, or inverting the value of a boolean.

```
+       Unary plus operator; indicates
        positive value (numbers are
        positive without this, however)
-       Unary minus operator; negates
        an expression
++      Increment operator; increments
        a value by 1
--      Decrement operator; decrements
        a value by 1
!       Logical complement operator;
        inverts the value of a boolean
```

The following program, `UnaryDemo`, tests the unary operators:

```
class UnaryDemo {

    public static void main(String[] args){
        // result is now 1
        int result = +1;
        System.out.println(result);
        // result is now 0
        result--;
        System.out.println(result);
        // result is now 1
        result++;
        System.out.println(result);
        // result is now -1
        result = -result;
        System.out.println(result);
        boolean success = false;
        // false
        System.out.println(success);
        // true
        System.out.println(!success);
    }
}
```

The increment/decrement operators can be applied before (prefix) or after (postfix) the operand. The code `result++;` and `++result;` will both end in `result` being incremented by one. The only difference is that the prefix version (`++result`) evaluates to the incremented value, whereas the postfix version (`result++`) evaluates to the original value. If you are just performing a simple increment/decrement, it doesn't really matter which version you choose. But if you use this operator in part of a larger expression, the one that you choose may make a significant difference.

The following program, `PrePostDemo`, illustrates the prefix/postfix unary increment operator:

```
class PrePostDemo {
    public static void main(String[] args){
        int i = 3;
        i++;
        // prints 4
        System.out.println(i);
        ++i;
        // prints 5
        System.out.println(i);
        // prints 6
        System.out.println(++i);
        // prints 6
        System.out.println(i++);
        // prints 7
        System.out.println(i);
    }
}
```

Equality, Relational, and Conditional Operators

The Equality and Relational Operators

The equality and relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand. The majority of these operators will probably look familiar to you as well. Keep in mind that you must use "==", not "=", when testing if two primitive values are equal.

```
==      equal to
!=      not equal to
>       greater than
>=      greater than or equal to
<       less than
<=      less than or equal to
```

The following program, `ComparisonDemo`, tests the comparison operators:

```


class ComparisonDemo {

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        if(value1 == value2)
            System.out.println("value1 == value2");
        if(value1 != value2)
            System.out.println("value1 != value2");
        if(value1 > value2)
            System.out.println("value1 > value2");
        if(value1 < value2)
            System.out.println("value1 < value2");
        if(value1 <= value2)
            System.out.println("value1 <= value2");
    }
}
```

Output:

```
value1 != value2
value1 <  value2
value1 <= value2
```

The Conditional Operators

The `&&` and `||` operators perform *Conditional-AND* and *Conditional-OR* operations on two boolean expressions. These operators exhibit "short-circuiting" behavior, which means that the second operand is evaluated only if needed.

```
&& Conditional-AND
|| Conditional-OR
```

The following program, `ConditionalDemo1`, tests these operators:

```
class ConditionalDemo1 {

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        if((value1 == 1) && (value2 == 2))
            System.out.println("value1 is 1 AND value2 is 2");
        if((value1 == 1) || (value2 == 1))
            System.out.println("value1 is 1 OR value2 is 1");
    }
}
```

Another conditional operator is `?:`, which can be thought of as shorthand for an `if-then-else` statement (discussed in the Control Flow Statements section of this lesson). This operator is also known as the *ternary operator* because it uses three operands. In the following example, this operator should be read as: "If `someCondition` is `true`, assign the value of `value1` to `result`. Otherwise, assign the value of `value2` to `result`."

The following program, `ConditionalDemo2`, tests the `?:` operator:

```
class ConditionalDemo2 {

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        int result;
        boolean someCondition = true;
        result = someCondition ? value1 : value2;

        System.out.println(result);
    }
}
```

Because `someCondition` is true, this program prints "1" to the screen. Use the `?:` operator instead of an `if-then-else` statement if it makes your code more readable; for example, when the expressions are compact and without side-effects (such as assignments).

The Type Comparison Operator instanceof

The `instanceof` operator compares an object to a specified type. You can use it to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.

The following program, `InstanceofDemo`, defines a parent class (named `Parent`), a simple interface (named `MyInterface`), and a child class (named `Child`) that inherits from the parent and implements the interface.

```
class InstanceofDemo {
    public static void main(String[] args) {
        Parent obj1 = new Parent();
        Parent obj2 = new Child();

        System.out.println("obj1 instanceof Parent: "
            + (obj1 instanceof Parent));
        System.out.println("obj1 instanceof Child: "
            + (obj1 instanceof Child));
        System.out.println("obj1 instanceof MyInterface: "
            + (obj1 instanceof MyInterface));
        System.out.println("obj2 instanceof Parent: "
            + (obj2 instanceof Parent));
        System.out.println("obj2 instanceof Child: "
            + (obj2 instanceof Child));
        System.out.println("obj2 instanceof MyInterface: "
            + (obj2 instanceof MyInterface));
    }
}

class Parent {}
class Child extends Parent implements MyInterface {}
interface MyInterface {}
```

Output:

```
obj1 instanceof Parent: true
obj1 instanceof Child: false
obj1 instanceof MyInterface: false
obj2 instanceof Parent: true
obj2 instanceof Child: true
obj2 instanceof MyInterface: true
```

When using the `instanceof` operator, keep in mind that `null` is not an instance of anything.

Bitwise and Bit Shift Operators

The Java programming language also provides operators that perform bitwise and bit shift operations on integral types. The operators discussed in this section are less commonly used. Therefore, their coverage is brief; the intent is to simply make you aware that these operators exist.

The unary bitwise complement operator "~" inverts a bit pattern; it can be applied to any of the integral types, making every "0" a "1" and every "1" a "0". For example, a `byte` contains 8 bits; applying this operator to a value whose bit pattern is "00000000" would change its pattern to "11111111".

The signed left shift operator "<<" shifts a bit pattern to the left, and the signed right shift operator ">>" shifts a bit pattern to the right. The bit pattern is given by the left-hand operand, and the number of positions to shift by the right-hand operand. The unsigned right shift operator ">>>" shifts a zero into the leftmost position, while the leftmost position after `">>"` depends on sign extension.

The bitwise `&` operator performs a bitwise AND operation.

The bitwise `^` operator performs a bitwise exclusive OR operation.

The bitwise `|` operator performs a bitwise inclusive OR operation.

The following program, `BitDemo`, uses the bitwise AND operator to print the number "2" to standard output.

```
class BitDemo {
    public static void main(String[] args) {
        int bitmask = 0x000F;
        int val = 0x2222;
        // prints "2"
        System.out.println(val & bitmask);
    }
}
```

Summary of Operators

The following quick reference summarizes the operators supported by the Java programming language.

Simple Assignment Operator

```
=       Simple assignment operator
```

Arithmetic Operators

```
+       Additive operator (also used
        for String concatenation)
-       Subtraction operator
*       Multiplication operator
/       Division operator
%       Remainder operator
```

Unary Operators

```
+       Unary plus operator; indicates
        positive value (numbers are
        positive without this, however)
-       Unary minus operator; negates
        an expression
++      Increment operator; increments
        a value by 1
--      Decrement operator; decrements
        a value by 1
!       Logical complement operator;
        inverts the value of a boolean
```

Equality and Relational Operators

```
==      Equal to
!=      Not equal to
>       Greater than
>=      Greater than or equal to
<       Less than
<=      Less than or equal to
```

Conditional Operators

```
&&      Conditional-AND
||      Conditional-OR
?:      Ternary (shorthand for
        if-then-else statement)
```

Type Comparison Operator

```
instanceof      Compares an object to
                a specified type
```

Bitwise and Bit Shift Operators

```
~       Unary bitwise complement
<<      Signed left shift
>>      Signed right shift
>>>     Unsigned right shift
&       Bitwise AND
^       Bitwise exclusive OR
|       Bitwise inclusive OR
```

Questions and Exercises: Operators

Questions

1. Consider the following code snippet.

```
arrayOfInts[j] > arrayOfInts[j+1]
```

Which operators does the code contain?

2. Consider the following code snippet.

```
int i = 10;
int n = i++%5;
```

a. What are the values of `i` and `n` after the code is executed?

b. What are the final values of `i` and `n` if instead of using the postfix increment operator (`i++`), you use the prefix version (`++i`))?

3. To invert the value of a `boolean`, which operator would you use?

4. Which operator is used to compare two values, `=` or `==` ?

5. Explain the following code sample: `result = someCondition ? value1 : value2;`

Exercises

1. Change the following program to use compound assignments:

```
class ArithmeticDemo {

     public static void main (String[] args){

          int result = 1 + 2; // result is now 3
          System.out.println(result);

          result = result - 1; // result is now 2
          System.out.println(result);

          result = result * 2; // result is now 4
          System.out.println(result);

          result = result / 2; // result is now 2
          System.out.println(result);

          result = result + 8; // result is now 10
          result = result % 7; // result is now 3
          System.out.println(result);
     }
}
```

2. In the following program, explain why the value "6" is printed twice in a row:

```
class PrePostDemo {
    public static void main(String[] args){
        int i = 3;
        i++;
        System.out.println(i);    // "4"
        ++i;
        System.out.println(i);    // "5"
        System.out.println(++i);  // "6"
        System.out.println(i++);  // "6"
        System.out.println(i);    // "7"
    }
}
```

Check your answers

Answers to Questions

1. Consider the following code snippet:

```
arrayOfInts[j] > arrayOfInts[j+1]
```

Question: What operators does the code contain?
Answer: `>`, `+`

2. Consider the following code snippet:

```
int i = 10;
int n = i++%5;
```

a. Question: What are the values of `i` and `n` after the code is executed?
Answer: `i` is 11, and `n` is 0.

b. Question: What are the final values of `i` and `n` if instead of using the postfix increment operator (`i++`), you use the prefix version (`++i)`)?
Answer: `i` is 11, and `n` is 1.

3. Question: To invert the value of a `boolean`, which operator would you use?
Answer: The logical complement operator "!".
4. Question: Which operator is used to compare two values, `=` or `==` ?
Answer: The `==` operator is used for comparison, and `=` is used for assignment.
5. Question: Explain the following code sample: `result = someCondition ? value1 : value2;`
Answer: This code should be read as: "If `someCondition` is `true`, assign the value of `value1` to `result`. Otherwise, assign the value of `value2` to `result`."

Exercises

1. Change the following program to use compound assignments:

```
class ArithmeticDemo {

    public static void main (String[] args){

        int result = 1 + 2; // result is now 3
        System.out.println(result);

        result = result - 1; // result is now 2
        System.out.println(result);

        result = result * 2; // result is now 4
        System.out.println(result);

        result = result / 2; // result is now 2
        System.out.println(result);

        result = result + 8; // result is now 10
        result = result % 7; // result is now 3
        System.out.println(result);

    }
}
```

Here is one solution:

```
class ArithmeticDemo {

    public static void main (String[] args){
        int result = 3;
        System.out.println(result);

        result -= 1; // result is now 2
        System.out.println(result);

        result *= 2; // result is now 4
        System.out.println(result);

        result /= 2; // result is now 2
        System.out.println(result);

        result += 8; // result is now 10
        result %= 7; // result is now 3
        System.out.println(result);

    }
}
```

2. In the following program, explain why the value "6" is printed twice in a row:

```
class PrePostDemo {
    public static void main(String[] args){
        int i = 3;
        i++;
        System.out.println(i);    // "4"
        ++i;
        System.out.println(i);    // "5"
        System.out.println(++i);  // "6"
        System.out.println(i++);  // "6"
        System.out.println(i);    // "7"
    }
}
```

The code `System.out.println(++i);` evaluates to 6, because the prefix version of `++` evaluates to the incremented value. The next line, `System.out.println(i++);` evaluates to the current value (6), then increments by one. So "7" doesn't get printed until the next line.

Expressions, Statements, and Blocks

Now that you understand variables and operators, it's time to learn about *expressions*, *statements*, and *blocks*. Operators may be used in building expressions, which compute values; expressions are the core components of statements; statements may be grouped into blocks.

Expressions

An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, that evaluates to a single value. You've already seen examples of expressions, illustrated in bold below:

```
int cadence = 0;
anArray[0] = 100;
System.out.println("Element 1 at index 0: " + anArray[0]);

int result = 1 + 2; // result is now 3
if (value1 == value2)
    System.out.println("value1 == value2");
```

The data type of the value returned by an expression depends on the elements used in the expression. The expression `cadence = 0` returns an `int` because the assignment operator returns a value of the same data type as its left-hand operand; in this case, `cadence` is an `int`. As you can see from the other expressions, an expression can return other types of values as well, such as `boolean` or `String`.

The Java programming language allows you to construct compound expressions from various smaller expressions as long as the data type required by one part of the expression matches the data type of the other. Here's an example of a compound expression:

```
1 * 2 * 3
```

In this particular example, the order in which the expression is evaluated is unimportant because the result of multiplication is independent of order; the outcome is always the same, no matter in which order you apply the multiplications. However, this is not true of all expressions. For example, the following expression gives different results, depending on whether you perform the addition or the division operation first:

```
x + y / 100    // ambiguous
```

You can specify exactly how an expression will be evaluated using balanced parenthesis: (and). For example, to make the previous expression unambiguous, you could write the following:

```
(x + y) / 100  // unambiguous, recommended
```

If you don't explicitly indicate the order for the operations to be performed, the order is determined by the precedence assigned to the operators in use within the expression. Operators that have a higher precedence get evaluated first. For example, the division operator has a higher precedence than does the addition operator. Therefore, the following two statements are equivalent:

```
x + y / 100


x + (y / 100) // unambiguous, recommended
```

When writing compound expressions, be explicit and indicate with parentheses which operators should be evaluated first. This practice makes code easier to read and to maintain.

Statements

Statements are roughly equivalent to sentences in natural languages. A *statement* forms a complete unit of execution. The following types of expressions can be made into a statement by terminating the expression with a semicolon (`;`).

- Assignment expressions
- Any use of `++` or `--`
- Method invocations
- Object creation expressions

Such statements are called *expression statements*. Here are some examples of expression statements.

```
// assignment statement
aValue = 8933.234;
// increment statement
aValue++;
// method invocation statement
System.out.println("Hello World!");
// object creation statement
Bicycle myBike = new Bicycle();
```

In addition to expression statements, there are two other kinds of statements: *declaration statements* and *control flow statements*. A *declaration statement* declares a variable. You've seen many examples of declaration statements already:

```
// declaration statement
double aValue = 8933.234;
```

Finally, *control flow statements* regulate the order in which statements get executed. You'll learn about control flow statements in the next section, Control Flow Statements

Blocks

A *block* is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed. The following example, `BlockDemo`, illustrates the use of blocks:

```
class BlockDemo {
     public static void main(String[] args) {
          boolean condition = true;
          if (condition) { // begin block 1
               System.out.println("Condition is true.");
          } // end block one
          else { // begin block 2
               System.out.println("Condition is false.");
          } // end block 2
     }
}
```

Questions and Exercises: Expressions, Statements, and Blocks

Questions

1. Operators may be used in building ___, which compute values.
2. Expressions are the core components of ___.
3. Statements may be grouped into ___.
4. The following code snippet is an example of a ___ expression.

```
1 * 2 * 3
```

5. Statements are roughly equivalent to sentences in natural languages, but instead of ending with a period, a statement ends with a ___.
6. A block is a group of zero or more statements between balanced ___ and can be used anywhere a single statement is allowed.

Exercises

Identify the following kinds of expression statements:

- `aValue = 8933.234;`
- `aValue++;`
- `System.out.println("Hello World!");`
- `Bicycle myBike = new Bicycle();`

Check your answers

Questions

1. Operators may be used in building **expressions**, which compute values.
2. Expressions are the core components of **statements**.
3. Statements may be grouped into **blocks**.
4. The following code snippet is an example of a **compound** expression.

```
1 * 2 * 3
```

5. Statements are roughly equivalent to sentences in natural languages, but instead of ending with a period, a statement ends with a **semicolon**.
6. A block is a group of zero or more statements between balanced **braces** and can be used anywhere a single statement is allowed.

Exercises

Identify the following kinds of expression statements:

- `aValue = 8933.234;` **`// assignment statement`**
- `aValue++; //` **`increment statement`**
- `System.out.println("Hello World!");` **`// method invocation statement`**
- `Bicycle myBike = new Bicycle();` **`// object creation statement`**

Control Flow Statements

The statements inside your source files are generally executed from top to bottom, in the order that they appear. *Control flow statements*, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to *conditionally* execute particular blocks of code. This section describes the decision-making statements (`if-then`, `if-then-else`, `switch`), the looping statements (`for`, `while`, `do-while`), and the branching statements (`break`, `continue`, `return`) supported by the Java programming language.

The if-then and if-then-else Statements

The `if-then` Statement

The `if-then` statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code *only if* a particular test evaluates to `true`. For example, the `Bicycle` class could allow the brakes to decrease the bicycle's speed *only if* the bicycle is already in motion. One possible implementation of the `applyBrakes` method could be as follows:

```
void applyBrakes() {
    // the "if" clause: bicycle must be moving
    if (isMoving){
        // the "then" clause: decrease current speed
        currentSpeed--;
    }
}
```

If this test evaluates to `false` (meaning that the bicycle is not in motion), control jumps to the end of the `if-then` statement.

In addition, the opening and closing braces are optional, provided that the "then" clause contains only one statement:

```
void applyBrakes() {
    // same as above, but without braces
    if (isMoving)
        currentSpeed--;
}
```

Deciding when to omit the braces is a matter of personal taste. Omitting them can make the code more brittle. If a second statement is later added to the "then" clause, a common mistake would be forgetting to add the newly required braces. The compiler cannot catch this sort of error; you'll just get the wrong results.

The `if-then-else` Statement

The `if-then-else` statement provides a secondary path of execution when an "if" clause evaluates to `false`. You could use an `if-then-else` statement in the `applyBrakes` method to take some action if the brakes are applied when the bicycle is not in motion. In this case, the action is to simply print an error message stating that the bicycle has already stopped.

```
void applyBrakes() {
    if (isMoving) {
        currentSpeed--;
    } else {
        System.err.println("The bicycle has already stopped!");
    }
}
```

The following program, `IfElseDemo`, assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on.

```
class IfElseDemo {
    public static void main(String[] args) {

        int testscore = 76;
        char grade;

        if (testscore >= 90) {
            grade = 'A';
        } else if (testscore >= 80) {
            grade = 'B';
        } else if (testscore >= 70) {
            grade = 'C';
        } else if (testscore >= 60) {
            grade = 'D';
        } else {
            grade = 'F';
        }
        System.out.println("Grade = " + grade);
    }
}
```

The output from the program is:

```
    Grade = C
```

You may have noticed that the value of `testscore` can satisfy more than one expression in the compound statement: `76 >= 70` and `76 >= 60`. However, once a condition is satisfied, the appropriate statements are executed `(grade = 'C';)` and the remaining conditions are not evaluated.

The switch Statement

Unlike `if-then` and `if-then-else` statements, the `switch` statement can have a number of possible execution paths. A `switch` works with the `byte`, `short`, `char`, and `int` primitive data types. It also works with *enumerated types* (discussed in Enum Types), the `String` class, and a few special classes that wrap certain primitive types: `Character`, `Byte`, `Short`, and `Integer` (discussed in Numbers and Strings).

The following code example, `SwitchDemo`, declares an `int` named `month` whose value represents a month. The code displays the name of the month, based on the value of `month`, using the `switch` statement.

```
public class SwitchDemo {
    public static void main(String[] args) {

        int month = 8;
        String monthString;
        switch (month) {
            case 1:  monthString = "January";
                     break;
            case 2:  monthString = "February";
                     break;
            case 3:  monthString = "March";
                     break;
            case 4:  monthString = "April";
                     break;
            case 5:  monthString = "May";
                     break;
            case 6:  monthString = "June";
                     break;
            case 7:  monthString = "July";
                     break;
            case 8:  monthString = "August";
                     break;
            case 9:  monthString = "September";
                     break;
            case 10: monthString = "October";
                     break;
            case 11: monthString = "November";
                     break;
            case 12: monthString = "December";
                     break;
            default: monthString = "Invalid month";
                     break;
        }
        System.out.println(monthString);
    }
}
```

In this case, `August` is printed to standard output.

The body of a `switch` statement is known as a *switch block*. A statement in the `switch` block can be labeled with one or more `case` or `default` labels. The `switch` statement evaluates its expression, then executes all statements that follow the matching `case` label.

You could also display the name of the month with `if-then-else` statements:

```
int month = 8;
if (month == 1) {
    System.out.println("January");
} else if (month == 2) {
    System.out.println("February");
}
...  // and so on
```

Deciding whether to use `if-then-else` statements or a `switch` statement is based on readability and the expression that the statement is testing. An `if-then-else` statement can test expressions based on ranges of values or conditions, whereas a `switch` statement tests expressions based only on a single integer, enumerated value, or `String` object.

Another point of interest is the `break` statement. Each `break` statement terminates the enclosing `switch` statement. Control flow continues with the first statement following the `switch` block. The `break` statements are necessary because without them, statements in `switch` blocks *fall through*: All statements after the matching `case` label are executed in sequence, regardless of the expression of subsequent `case` labels, until a `break` statement is encountered. The program `SwitchDemoFallThrough` shows statements in a `switch` block that fall through. The program displays the month corresponding to the integer `month` and the months that follow in the year:

```
public class SwitchDemoFallThrough {

    public static void main(String[] args) {
        java.util.ArrayList<String> futureMonths =
            new java.util.ArrayList<String>();

        int month = 8;

        switch (month) {
            case 1:  futureMonths.add("January");
            case 2:  futureMonths.add("February");
            case 3:  futureMonths.add("March");
            case 4:  futureMonths.add("April");
            case 5:  futureMonths.add("May");
            case 6:  futureMonths.add("June");
            case 7:  futureMonths.add("July");
            case 8:  futureMonths.add("August");
            case 9:  futureMonths.add("September");
            case 10: futureMonths.add("October");
            case 11: futureMonths.add("November");
            case 12: futureMonths.add("December");
                     break;
            default: break;
        }

        if (futureMonths.isEmpty()) {
            System.out.println("Invalid month number");
        } else {
            for (String monthName : futureMonths) {
               System.out.println(monthName);
            }
        }
    }
}
```

This is the output from the code:

```
August
September
October
November
December
```

Technically, the final `break` is not required because flow falls out of the `switch` statement. Using a `break` is recommended so that modifying the code is easier and less error prone. The `default` section handles all values that are not explicitly handled by one of the `case` sections.

The following code example, `SwitchDemo2`, shows how a statement can have multiple `case` labels. The code example calculates the number of days in a particular month:

```
class SwitchDemo2 {
    public static void main(String[] args) {

        int month = 2;
        int year = 2000;
        int numDays = 0;

        switch (month) {
            case 1: case 3: case 5:
            case 7: case 8: case 10:
            case 12:
                numDays = 31;
                break;
            case 4: case 6:
            case 9: case 11:
                numDays = 30;
                break;
            case 2:
                if (((year % 4 == 0) &&
                     !(year % 100 == 0))
                     || (year % 400 == 0))
                    numDays = 29;
                else
                    numDays = 28;
                break;
            default:
                System.out.println("Invalid month.");
                break;
        }
        System.out.println("Number of Days = "
                           + numDays);
    }
}
```

This is the output from the code:

```
Number of Days = 29
```

Using Strings in switch Statements

In Java SE 7 and later, you can use a `String` object in the `switch` statement's expression. The following code example, `StringSwitchDemo`, displays the number of the month based on the value of the `String` named `month`:

```
public class StringSwitchDemo {

    public static int getMonthNumber(String month) {

        int monthNumber = 0;

        if (month == null) {
            return monthNumber;
        }

        switch (month.toLowerCase()) {
            case "january":
                monthNumber = 1;
                break;
            case "february":
                monthNumber = 2;
                break;
            case "march":
                monthNumber = 3;
                break;
            case "april":
                monthNumber = 4;
                break;
            case "may":
                monthNumber = 5;
                break;
            case "june":
                monthNumber = 6;
                break;
            case "july":
                monthNumber = 7;
                break;
            case "august":
                monthNumber = 8;
                break;
            case "september":
                monthNumber = 9;
                break;
            case "october":
                monthNumber = 10;
                break;
            case "november":
                monthNumber = 11;
                break;
            case "december":
                monthNumber = 12;
                break;
            default:
                monthNumber = 0;
                break;
        }

        return monthNumber;
    }

    public static void main(String[] args) {

        String month = "August";

        int returnedMonthNumber =
            StringSwitchDemo.getMonthNumber(month);

        if (returnedMonthNumber == 0) {
            System.out.println("Invalid month");
        } else {
            System.out.println(returnedMonthNumber);
        }
    }
}
```

The output from this code is `8`.

The `String` in the `switch` expression is compared with the expressions associated with each `case` label as if the [`String.equals`]() method were being used. In order for the `StringSwitchDemo` example to accept any month regardless of case, `month` is converted to lowercase (with the [`toLowerCase`]() method), and all the strings associated with the `case` labels are in lowercase.

Note: This example checks if the expression in the `switch` statement is `null`. Ensure that the expression in any `switch` statement is not null to prevent a `NullPointerException` from being thrown.

The while and do-while Statements

The `while` statement continually executes a block of statements while a particular condition is `true`. Its syntax can be expressed as:

```
while (expression) {
     statement(s)
}
```

The `while` statement evaluates *expression*, which must return a `boolean` value. If the expression evaluates to `true`, the `while` statement executes the *statement*(s) in the `while` block. The `while` statement continues testing the expression and executing its block until the expression evaluates to `false`. Using the `while` statement to print the values from 1 through 10 can be accomplished as in the following `WhileDemo` program:

```
class WhileDemo {
    public static void main(String[] args){
        int count = 1;
        while (count < 11) {
            System.out.println("Count is: " + count);
            count++;
        }
    }
}
```

You can implement an infinite loop using the `while` statement as follows:

```
while (true){
    // your code goes here
}
```

The Java programming language also provides a `do-while` statement, which can be expressed as follows:

```
do {
     statement(s)
} while (expression);
```

The difference between `do-while` and `while` is that `do-while` evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the `do` block are always executed at least once, as shown in the following `DoWhileDemo` program:

```
class DoWhileDemo {
    public static void main(String[] args){
        int count = 1;
        do {
            System.out.println("Count is: " + count);
            count++;
        } while (count < 11);
    }
}
```

The for Statement

The `for` statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the `for` statement can be expressed as follows:

```
for (initialization; termination;
     increment) {
    statement(s)
}
```

When using this version of the `for` statement, keep in mind that:

- The *initialization* expression initializes the loop; it's executed once, as the loop begins.
- When the *termination* expression evaluates to `false`, the loop terminates.
- The *increment* expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment *or* decrement a value.

The following program, `ForDemo`, uses the general form of the `for` statement to print the numbers 1 through 10 to standard output:

```
class ForDemo {
    public static void main(String[] args){
         for(int i=1; i<11; i++){
              System.out.println("Count is: " + i);
         }
    }
}
```

The output of this program is:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

Notice how the code declares a variable within the initialization expression. The scope of this variable extends from its declaration to the end of the block governed by the `for` statement, so it can be used in the termination and increment expressions as well. If the variable that controls a `for` statement is not needed outside of the loop, it's best to declare the variable in the initialization expression. The names `i`, `j`, and `k` are often used to control `for` loops; declaring them within the initialization expression limits their life span and reduces errors.

The three expressions of the `for` loop are optional; an infinite loop can be created as follows:

```
// infinite loop
for ( ; ; ) {

    // your code goes here
}
```

The `for` statement also has another form designed for iteration through Collections and arrays This form is sometimes referred to as the *enhanced for* statement, and can be used to make your loops more compact and easy to read. To demonstrate, consider the following array, which holds the numbers 1 through 10:

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};
```

The following program, `EnhancedForDemo`, uses the enhanced `for` to loop through the array:

```


class EnhancedForDemo {
    public static void main(String[] args){
         int[] numbers =
             {1,2,3,4,5,6,7,8,9,10};
         for (int item : numbers) {
             System.out.println("Count is: " + item);
         }
    }
}
```

In this example, the variable `item` holds the current value from the numbers array. The output from this program is the same as before:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

We recommend using this form of the `for` statement instead of the general form whenever possible.

Branching Statements

The `break` Statement

The `break` statement has two forms: labeled and unlabeled. You saw the unlabeled form in the previous discussion of the `switch` statement. You can also use an unlabeled `break` to terminate a `for`, `while`, or `do-while` loop, as shown in the following `BreakDemo` program:

```
class BreakDemo {
    public static void main(String[] args) {

        int[] arrayOfInts =
            { 32, 87, 3, 589,
              12, 1076, 2000,
              8, 622, 127 };
        int searchfor = 12;

        int i;
        boolean foundIt = false;

        for (i = 0; i < arrayOfInts.length; i++) {
            if (arrayOfInts[i] == searchfor) {
                foundIt = true;
                break;
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor + " at index " + i);
        } else {
            System.out.println(searchfor + " not in the array");
        }
    }
}
```

This program searches for the number 12 in an array. The `break` statement, shown in boldface, terminates the `for` loop when that value is found. Control flow then transfers to the statement after the `for` loop. This program's output is:

```
Found 12 at index 4
```

An unlabeled `break` statement terminates the innermost `switch`, `for`, `while`, or `do-while` statement, but a labeled `break` terminates an outer statement. The following program, `BreakWithLabelDemo`, is similar to the previous program, but uses nested `for` loops to search for a value in a two-dimensional array. When the value is found, a labeled `break` terminates the outer `for` loop (labeled "search"):

```
class BreakWithLabelDemo {
    public static void main(String[] args) {

        int[][] arrayOfInts = {
            { 32, 87, 3, 589 },
            { 12, 1076, 2000, 8 },
            { 622, 127, 77, 955 }
        };
        int searchfor = 12;
        int i;
        int j = 0;
        boolean foundIt = false;

    search:
        for (i = 0; i < arrayOfInts.length; i++) {
            for (j = 0; j < arrayOfInts[i].length;
                 j++) {
                if (arrayOfInts[i][j] == searchfor) {
                    foundIt = true;
                    break search;
                }
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor + " at " + i + ", " + j);
        } else {
            System.out.println(searchfor + " not in the array");
        }
    }
}
```

This is the output of the program.

```
Found 12 at 1, 0
```

The `break` statement terminates the labeled statement; it does not transfer the flow of control to the label. Control flow is transferred to the statement immediately following the labeled (terminated) statement.

The `continue` Statement

The `continue` statement skips the current iteration of a `for`, `while` , or `do-while` loop. The unlabeled form skips to the end of the innermost loop's body and evaluates the `boolean` expression that controls the loop. The following program, `ContinueDemo` , steps through a `String`, counting the occurences of the letter "p". If the current character is not a p, the `continue` statement skips the rest of the loop and proceeds to the next character. If it *is* a "p", the program increments the letter count.

```
class ContinueDemo {
    public static void main(String[] args) {

        String searchMe = "peter piper picked a " + "peck of pickled peppers";
        int max = searchMe.length();
        int numPs = 0;

        for (int i = 0; i < max; i++) {
            // interested only in p's
            if (searchMe.charAt(i) != 'p')
                continue;

            // process p's
            numPs++;
        }
        System.out.println("Found " + numPs + " p's in the string.");
    }
}
```

Here is the output of this program:

```
Found 9 p's in the string.
```

To see this effect more clearly, try removing the `continue` statement and recompiling. When you run the program again, the count will be wrong, saying that it found 35 p's instead of 9.

A labeled `continue` statement skips the current iteration of an outer loop marked with the given label. The following example program, `ContinueWithLabelDemo`, uses nested loops to search for a substring within another string. Two nested loops are required: one to iterate over the substring and one to iterate over the string being searched. The following program, `ContinueWithLabelDemo`, uses the labeled form of continue to skip an iteration in the outer loop.

```
class ContinueWithLabelDemo {
    public static void main(String[] args) {

        String searchMe = "Look for a substring in me";
        String substring = "sub";
        boolean foundIt = false;

        int max = searchMe.length() - 
                  substring.length();

    test:
        for (int i = 0; i <= max; i++) {
            int n = substring.length();
            int j = i;
            int k = 0;
            while (n-- != 0) {
                if (searchMe.charAt(j++) != substring.charAt(k++)) {
                    continue test;
                }
            }
            foundIt = true;
                break test;
        }
        System.out.println(foundIt ? "Found it" : "Didn't find it");
    }
}
```

Here is the output from this program.

```
Found it
```

The `return` Statement

The last of the branching statements is the `return` statement. The `return` statement exits from the current method, and control flow returns to where the method was invoked. The `return` statement has two forms: one that returns a value, and one that doesn't. To return a value, simply put the value (or an expression that calculates the value) after the `return` keyword.

```
return ++count;
```

The data type of the returned value must match the type of the method's declared return value. When a method is declared `void`, use the form of `return` that doesn't return a value.

```
return;
```

The Classes and Objects lesson will cover everything you need to know about writing methods.

Summary of Control Flow Statements

The `if-then` statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code *only if* a particular test evaluates to `true`. The `if-then-else` statement provides a secondary path of execution when an "if" clause evaluates to `false`. Unlike `if-then` and `if-then-else`, the `switch` statement allows for any number of possible execution paths. The `while` and `do-while` statements continually execute a block of statements while a particular condition is `true`. The difference between `do-while` and `while` is that `do-while` evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the `do` block are always executed at least once. The `for` statement provides a compact way to iterate over a range of values. It has two forms, one of which was designed for looping through collections and arrays.

Questions and Exercises: Control Flow Statements

Questions

1. The most basic control flow statement supported by the Java programming language is the ___ statement.
2. The ___ statement allows for any number of possible execution paths.
3. The ___ statement is similar to the `while` statement, but evaluates its expression at the ___ of the loop.
4. How do you write an infinite loop using the `for` statement?
5. How do you write an infinite loop using the `while` statement?

Exercises

1. Consider the following code snippet.

```
if (aNumber >= 0)
    if (aNumber == 0)
        System.out.println("first string");
else System.out.println("second string");
System.out.println("third string");
```

a. What output do you think the code will produce if `aNumber` is 3?
b. Write a test program containing the previous code snippet; make `aNumber` 3. What is the output of the program? Is it what you predicted? Explain why the output is what it is; in other words, what is the control flow for the code snippet?
c. Using only spaces and line breaks, reformat the code snippet to make the control flow easier to understand.
d. Use braces, { and }, to further clarify the code.

Check your answers

Answers to Questions

1. The most basic control flow statement supported by the Java programming language is the **if-then** statement.
2. The **switch** statement allows for any number of possible execution paths.
3. The **do-while** statement is similar to the `while` statement, but evaluates its expression at the **bottom** of the loop.
4. Question: How do you write an infinite loop using the `for` statement? **Answer:**

```
for ( ; ; ) {

}
```

5. Question: How do you write an infinite loop using the `while` statement? **Answer:**

```
while (true) {

}
```

Exercises

1. Consider the following code snippet.

```
if (aNumber >= 0)
    if (aNumber == 0)
        System.out.println("first string");
else
    System.out.println("second string");
System.out.println("third string");
```

a. Exercise: What output do you think the code will produce if `aNumber` is 3? **Solution:**

```
second string
third string
```

b. Exercise: Write a test program containing the previous code snippet; make `aNumber` 3. What is the output of the program? Is it what you predicted? Explain why the output is what it is. In other words, what is the control flow for the code snippet? **Solution:** `NestedIf`

```
second string
third string
```

3 is greater than or equal to 0, so execution progresses to the second `if` statement. The second `if` statement's test fails because 3 is not equal to 0. Thus, the `else` clause executes (since it's attached to the second `if` statement). Thus, `second string` is displayed. The final `println` is completely outside of any `if` statement, so it always gets executed, and thus `third string` is always displayed.
c. Exercise: Using only spaces and line breaks, reformat the code snippet to make the control flow easier to understand. **Solution:**

```
if (aNumber >= 0)
    if (aNumber == 0)
        System.out.println("first string");
    else
        System.out.println("second string");

System.out.println("third string");
```

d. Exercise: Use braces `{` and `}` to further clarify the code and reduce the possibility of errors by future maintainers of the code. **Solution:**

```
if (aNumber >= 0) {
    if (aNumber == 0) {
        System.out.println("first string");
    } else {
        System.out.println("second string");
    }
}

System.out.println("third string");
```

Lesson: Classes and Objects

With the knowledge you now have of the basics of the Java programming language, you can learn to write your own classes. In this lesson, you will find information about defining your own classes, including declaring member variables, methods, and constructors.

You will learn to use your classes to create objects, and how to use the objects you create.

This lesson also covers nesting classes within other classes, and enumerations

Classes

This section shows you the anatomy of a class, and how to declare fields, methods, and constructors.

Objects

This section covers creating and using objects. You will learn how to instantiate an object, and, once instantiated, how to use the `dot` operator to access the object's instance variables and methods.

More on Classes

This section covers more aspects of classes that depend on using object references and the `dot` operator that you learned about in the preceding section: returning values from methods, the `this` keyword, class vs. instance members, and access control.

Nested Classes

Static nested classes, inner classes, anonymous inner classes, local classes, and lambda expressions are covered. There is also a discussion on when to use which approach.

Enum Types

This section covers enumerations, specialized classes that allow you to define and use sets of constants.

Note: See online version of topics in this ebook to download complete source code.

Classes

The introduction to object-oriented concepts in the lesson titled Object-oriented Programming Concepts used a bicycle class as an example, with racing bikes, mountain bikes, and tandem bikes as subclasses. Here is sample code for a possible implementation of a `Bicycle` class, to give you an overview of a class declaration. Subsequent sections of this lesson will back up and explain class declarations step by step. For the moment, don't concern yourself with the details.

```
public class Bicycle {

    // the Bicycle class has
    // three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has
    // one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has
    // four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

}
```

A class declaration for a `MountainBike` class that is a subclass of `Bicycle` might look like this:

```
public class MountainBike extends Bicycle {

    // the MountainBike subclass has
    // one field
    public int seatHeight;

    // the MountainBike subclass has
    // one constructor
    public MountainBike(int startHeight, int startCadence,
                        int startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }
    // the MountainBike subclass has
    // one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }

}
```

`MountainBike` inherits all the fields and methods of `Bicycle` and adds the field `seatHeight` and a method to set it (mountain bikes have seats that can be moved up and down as the terrain demands).

Declaring Classes

You've seen classes defined in the following way:

```
class MyClass {
    // field, constructor, and
    // method declarations
}
```

This is a *class declaration*. The *class body* (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class: constructors for initializing new objects, declarations for the fields that provide the state of the class and its objects, and methods to implement the behavior of the class and its objects.

The preceding class declaration is a minimal one. It contains only those components of a class declaration that are required. You can provide more information about the class, such as the name of its superclass, whether it implements any interfaces, and so on, at the start of the class declaration. For example,

```
class MyClass extends MySuperClass implements YourInterface {
    // field, constructor, and
    // method declarations
}
```

means that `MyClass` is a subclass of `MySuperClass` and that it implements the `YourInterface` interface.

You can also add modifiers like *public* or *private* at the very beginning—so you can see that the opening line of a class declaration can become quite complicated. The modifiers *public* and *private*, which determine what other classes can access `MyClass`, are discussed later in this lesson. The lesson on interfaces and inheritance will explain how and why you would use the *extends* and *implements* keywords in a class declaration. For the moment you do not need to worry about these extra complications.

In general, class declarations can include these components, in order:

1. Modifiers such as *public*, *private*, and a number of others that you will encounter later.
2. The class name, with the initial letter capitalized by convention.
3. The name of the class's parent (superclass), if any, preceded by the keyword *extends*. A class can only *extend* (subclass) one parent.
4. A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword *implements*. A class can *implement* more than one interface.
5. The class body, surrounded by braces, {}.

Declaring Member Variables

There are several kinds of variables:

- Member variables in a class—these are called *fields*.
- Variables in a method or block of code—these are called *local variables*.
- Variables in method declarations—these are called *parameters*.

The `Bicycle` class uses the following lines of code to define its fields:

```
public int cadence;
public int gear;
public int speed;
```

Field declarations are composed of three components, in order:

1. Zero or more modifiers, such as `public` or `private`.
2. The field's type.
3. The field's name.

The fields of `Bicycle` are named `cadence`, `gear`, and `speed` and are all of data type integer (`int`). The `public` keyword identifies these fields as public members, accessible by any object that can access the class.

Access Modifiers

The first (left-most) modifier used lets you control what other classes have access to a member field. For the moment, consider only `public` and `private`. Other access modifiers will be discussed later.

- `public` modifier—the field is accessible from all classes.
- `private` modifier—the field is accessible only within its own class.

In the spirit of encapsulation, it is common to make fields private. This means that they can only be *directly* accessed from the Bicycle class. We still need access to these values, however. This can be done *indirectly* by adding public methods that obtain the field values for us:

```
public class Bicycle {

    private int cadence;
    private int gear;
    private int speed;

    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    public int getCadence() {
        return cadence;
    }
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public int getGear() {
        return gear;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public int getSpeed() {
        return speed;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }
}
```

Types

All variables must have a type. You can use primitive types such as `int`, `float`, `boolean`, etc. Or you can use reference types, such as strings, arrays, or objects.

Variable Names

All variables, whether they are fields, local variables, or parameters, follow the same naming rules and conventions that were covered in the Language Basics lesson, Variables—Naming.

In this lesson, be aware that the same naming rules and conventions are used for method and class names, except that

- the first letter of a class name should be capitalized, and
- the first (or only) word in a method name should be a verb.

Defining Methods

Here is an example of a typical method declaration:

```
public double calculateAnswer(double wingSpan, int numberOfEngines,
                              double length, double grossTons) {
    //do the calculation here
}
```

The only required elements of a method declaration are the method's return type, name, a pair of parentheses, `()`, and a body between braces, `{}`.

More generally, method declarations have six components, in order:

1. Modifiers—such as `public`, `private`, and others you will learn about later.
2. The return type—the data type of the value returned by the method, or `void` if the method does not return a value.
3. The method name—the rules for field names apply to method names as well, but the convention is a little different.
4. The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, `()`. If there are no parameters, you must use empty parentheses.
5. An exception list—to be discussed later.
6. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

Modifiers, return types, and parameters will be discussed later in this lesson. Exceptions are discussed in a later lesson.

Definition: Two of the components of a method declaration comprise the *method signature*—the method's name and the parameter types.

The signature of the method declared above is:

```
calculateAnswer(double, int, double, double)
```

Naming a Method

Although a method name can be any legal identifier, code conventions restrict method names. By convention, method names should be a verb in lowercase or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc. In multi-word names, the first letter of each of the second and following words should be capitalized. Here are some examples:

```
run
runFast
getBackground
```

```
getFinalData
compareTo
setX
isEmpty
```

Typically, a method has a unique name within its class. However, a method might have the same name as other methods due to *method overloading*.

Overloading Methods

The Java programming language supports *overloading* methods, and Java can distinguish between methods with different *method signatures*. This means that methods within a class can have the same name if they have different parameter lists (there are some qualifications to this that will be discussed in the lesson titled "Interfaces and Inheritance").

Suppose that you have a class that can use calligraphy to draw various types of data (strings, integers, and so on) and that contains a method for drawing each data type. It is cumbersome to use a new name for each method—for example, `drawString`, `drawInteger`, `drawFloat`, and so on. In the Java programming language, you can use the same name for all the drawing methods but pass a different argument list to each method. Thus, the data drawing class might declare four methods named `draw`, each of which has a different parameter list.

```
public class DataArtist {
    ...
    public void draw(String s) {
        ...
    }
    public void draw(int i) {
        ...
    }
    public void draw(double f) {
        ...
    }
    public void draw(int i, double f) {
        ...
    }
}
```

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, `draw(String s)` and `draw(int i)` are distinct and unique methods because they require different argument types.

You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

Note: Overloaded methods should be used sparingly, as they can make code much less readable.

Providing Constructors for Your Classes

A class contains constructors that are invoked to create objects from the class blueprint. Constructor declarations look like method declarations—except that they use the name of the class and have no return type. For example, `Bicycle` has one constructor:

```
public Bicycle(int startCadence, int startSpeed, int startGear) {
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;
}
```

To create a new `Bicycle` object called `myBike`, a constructor is called by the `new` operator:

```
Bicycle myBike = new Bicycle(30, 0, 8);
```

`new Bicycle(30, 0, 8)` creates space in memory for the object and initializes its fields.

Although `Bicycle` only has one constructor, it could have others, including a no-argument constructor:

```
public Bicycle() {
    gear = 1;
    cadence = 10;
    speed = 0;
}
```

`Bicycle yourBike = new Bicycle();` invokes the no-argument constructor to create a new `Bicycle` object called `yourBike`.

Both constructors could have been declared in `Bicycle` because they have different argument lists. As with methods, the Java platform differentiates constructors on the basis of the number of arguments in the list and their types. You cannot write two constructors that have the same number and type of arguments for the same class, because the platform would not be able to tell them apart. Doing so causes a compile-time error.

You don't have to provide any constructors for your class, but you must be careful when doing this. The compiler automatically provides a no-argument, default constructor for any class without constructors. This default constructor will call the no-argument constructor of the superclass. In this situation, the compiler will complain if the superclass doesn't have a no-argument constructor so you must verify that it does. If your class has no explicit superclass, then it has an implicit superclass of `Object`, which *does* have a no-argument constructor.

You can use a superclass constructor yourself. The `MountainBike` class at the beginning of this lesson did just that. This will be discussed later, in the lesson on interfaces and inheritance.

You can use access modifiers in a constructor's declaration to control which other classes can call the constructor.

Note: If another class cannot call a `MyClass` constructor, it cannot directly create `MyClass` objects.

Passing Information to a Method or a Constructor

The declaration for a method or a constructor declares the number and the type of the arguments for that method or constructor. For example, the following is a method that computes the monthly payments for a home loan, based on the amount of the loan, the interest rate, the length of the loan (the number of periods), and the future value of the loan:

```
public double computePayment(
                  double loanAmt,
                  double rate,
                  double futureValue,
                  int numPeriods) {
    double interest = rate / 100.0;
    double partial1 = Math.pow((1 + interest),
                    - numPeriods);
    double denominator = (1 - partial1) / interest;
    double answer = (-loanAmt / denominator)
                    - ((futureValue * partial1) / denominator);
    return answer;
}
```

This method has four parameters: the loan amount, the interest rate, the future value and the number of periods. The first three are double-precision floating point numbers, and the fourth is an integer. The parameters are used in the method body and at runtime will take on the values of the arguments that are passed in.

Note:*Parameters* refers to the list of variables in a method declaration. *Arguments* are the actual values that are passed in when the method is invoked. When you invoke a method, the arguments used must match the declaration's parameters in type and order.

Parameter Types

You can use any data type for a parameter of a method or a constructor. This includes primitive data types, such as doubles, floats, and integers, as you saw in the `computePayment` method, and reference data types, such as objects and arrays.

Here's an example of a method that accepts an array as an argument. In this example, the method creates a new `Polygon` object and initializes it from an array of `Point` objects (assume that `Point` is a class that represents an x, y coordinate):

```
public Polygon polygonFrom(Point[] corners) {
    // method body goes here
}
```

Note: The Java programming language doesn't let you pass methods into methods. But you can pass an object into a method and then invoke the object's methods.

Arbitrary Number of Arguments

You can use a construct called *varargs* to pass an arbitrary number of values to a method. You use varargs when you don't know how many of a particular type of argument will be passed to the method. It's a shortcut to creating an array manually (the previous method could have used varargs rather than an array).

To use varargs, you follow the type of the last parameter by an ellipsis (three dots, ...), then a space, and the parameter name. The method can then be called with any number of that parameter, including none.

```
public Polygon polygonFrom(Point... corners) {
    int numberOfSides = corners.length;
    double squareOfSide1, lengthOfSide1;
    squareOfSide1 = (corners[1].x - corners[0].x)
                    * (corners[1].x - corners[0].x)
                    + (corners[1].y - corners[0].y)
                    * (corners[1].y - corners[0].y);
    lengthOfSide1 = Math.sqrt(squareOfSide1);

    // more method body code follows that creates and returns a
    // polygon connecting the Points
}
```

You can see that, inside the method, `corners` is treated like an array. The method can be called either with an array or with a sequence of arguments. The code in the method body will treat the parameter as an array in either case.

You will most commonly see varargs with the printing methods; for example, this `printf` method:

```
public PrintStream printf(String format, Object... args)
```

allows you to print an arbitrary number of objects. It can be called like this:

```
System.out.printf("%s: %d, %s%n", name, idnum, address);
```

or like this

```
System.out.printf("%s: %d, %s, %s, %s%n", name, idnum, address, phone, email);
```

or with yet a different number of arguments.

Parameter Names

When you declare a parameter to a method or a constructor, you provide a name for that parameter. This name is used within the method body to refer to the passed-in argument.

The name of a parameter must be unique in its scope. It cannot be the same as the name of another parameter for the same method or constructor, and it cannot be the name of a local variable within the method or constructor.

A parameter can have the same name as one of the class's fields. If this is the case, the parameter is said to *shadow* the field. Shadowing fields can make your code difficult to read and is conventionally used only within constructors and methods that set a particular field. For example, consider the following `Circle` class and its `setOrigin` method:

```
public class Circle {
    private int x, y, radius;
    public void setOrigin(int x, int y) {
        ...
    }
}
```

The `Circle` class has three fields: `x`, `y`, and `radius`. The `setOrigin` method has two parameters, each of which has the same name as one of the fields. Each method parameter shadows the field that shares its name. So using the simple names `x` or `y` within the body of the method refers to the parameter, *not* to the field. To access the field, you must use a qualified name. This will be discussed later in this lesson in the section titled "Using the `this` Keyword."

Passing Primitive Data Type Arguments

Primitive arguments, such as an `int` or a `double`, are passed into methods *by value*. This means that any changes to the values of the parameters exist only within the scope of the method. When the method returns, the parameters are gone and any changes to them are lost. Here is an example:

```
public class PassPrimitiveByValue {

    public static void main(String[] args) {

        int x = 3;

        // invoke passMethod() with
        // x as argument
        passMethod(x);

        // print x to see if its
        // value has changed
        System.out.println("After invoking passMethod, x = " + x);

    }

    // change parameter in passMethod()
    public static void passMethod(int p) {
        p = 10;
    }
}
```

When you run this program, the output is:

```
After invoking passMethod, x = 3
```

Passing Reference Data Type Arguments

Reference data type parameters, such as objects, are also passed into methods *by value*. This means that when the method returns, the passed-in reference still references the same object as before. *However*, the values of the object's fields *can* be changed in the method, if they have the proper access level.

For example, consider a method in an arbitrary class that moves `Circle` objects:

```
public void moveCircle(Circle circle, int deltaX, int deltaY) {
    // code to move origin of circle to x+deltaX, y+deltaY
    circle.setX(circle.getX() + deltaX);
    circle.setY(circle.getY() + deltaY);

    // code to assign a new reference to circle
    circle = new Circle(0, 0);
}
```

Let the method be invoked with these arguments:

```
moveCircle(myCircle, 23, 56)
```

Inside the method, `circle` initially refers to `myCircle`. The method changes the x and y coordinates of the object that `circle` references (i.e., `myCircle`) by 23 and 56, respectively. These changes will persist when the method returns. Then `circle` is assigned a reference to a new `Circle` object with `x = y = 0`. This reassignment has no permanence, however, because the reference was passed in by value and cannot change. Within the method, the object pointed to by `circle` has changed, but, when the method returns, `myCircle` still references the same `Circle` object as before the method was called.

Objects

A typical Java program creates many objects, which as you know, interact by invoking methods. Through these object interactions, a program can carry out various tasks, such as implementing a GUI, running an animation, or sending and receiving information over a network. Once an object has completed the work for which it was created, its resources are recycled for use by other objects.

Here's a small program, called `CreateObjectDemo`, that creates three objects: one `Point` object and two `Rectangle` objects. You will need all three source files to compile this program.

```
public class CreateObjectDemo {

    public static void main(String[] args) {

        // Declare and create a point object and two rectangle objects.
        Point originOne = new Point(23, 94);
        Rectangle rectOne = new Rectangle(originOne, 100, 200);
        Rectangle rectTwo = new Rectangle(50, 100);

        // display rectOne's width, height, and area
        System.out.println("Width of rectOne: " + rectOne.width);
        System.out.println("Height of rectOne: " + rectOne.height);
        System.out.println("Area of rectOne: " + rectOne.getArea());

        // set rectTwo's position
        rectTwo.origin = originOne;

        // display rectTwo's position
        System.out.println("X Position of rectTwo: " + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);

        // move rectTwo and display its new position
        rectTwo.move(40, 72);
        System.out.println("X Position of rectTwo: " + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);
    }
}
```

This program creates, manipulates, and displays information about various objects. Here's the output:

```
Width of rectOne: 100
Height of rectOne: 200
Area of rectOne: 20000
X Position of rectTwo: 23
Y Position of rectTwo: 94
X Position of rectTwo: 40
Y Position of rectTwo: 72
```

The following three sections use the above example to describe the life cycle of an object within a program. From them, you will learn how to write code that creates and uses objects in your own programs. You will also learn how the system cleans up after an object when its life has ended.

Creating Objects

As you know, a class provides the blueprint for objects; you create an object from a class. Each of the following statements taken from the `CreateObjectDemo` program creates an object and assigns it to a variable:

```
Point originOne = new Point(23, 94);
Rectangle rectOne = new Rectangle(originOne, 100, 200);
Rectangle rectTwo = new Rectangle(50, 100);
```

The first line creates an object of the `Point` class, and the second and third lines each create an object of the `Rectangle` class.

Each of these statements has three parts (discussed in detail below):

1. Declaration: The code set in **bold** are all variable declarations that associate a variable name with an object type.
2. Instantiation: The `new` keyword is a Java operator that creates the object.
3. Initialization: The `new` operator is followed by a call to a constructor, which initializes the new object.

Declaring a Variable to Refer to an Object

Previously, you learned that to declare a variable, you write:

```
type name;
```

This notifies the compiler that you will use *name* to refer to data whose type is *type*. With a primitive variable, this declaration also reserves the proper amount of memory for the variable.

You can also declare a reference variable on its own line. For example:

```
Point originOne;
```

If you declare `originOne` like this, its value will be undetermined until an object is actually created and assigned to it. Simply declaring a reference variable does not create an object. For that, you need to use the `new` operator, as described in the next section. You must assign an object to `originOne` before you use it in your code. Otherwise, you will get a compiler error.

A variable in this state, which currently references no object, can be illustrated as follows (the variable name, `originOne`, plus a reference pointing to nothing):

Instantiating a Class

The `new` operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The `new` operator also invokes the object constructor.

Note: The phrase "instantiating a class" means the same thing as "creating an object." When you create an object, you are creating an "instance" of a class, therefore "instantiating" a class.

The `new` operator requires a single, postfix argument: a call to a constructor. The name of the constructor provides the name of the class to instantiate.

The `new` operator returns a reference to the object it created. This reference is usually assigned to a variable of the appropriate type, like:

```
Point originOne = new Point(23, 94);
```

The reference returned by the `new` operator does not have to be assigned to a variable. It can also be used directly in an expression. For example:

```
int height = new Rectangle().height;
```

This statement will be discussed in the next section.

Initializing an Object

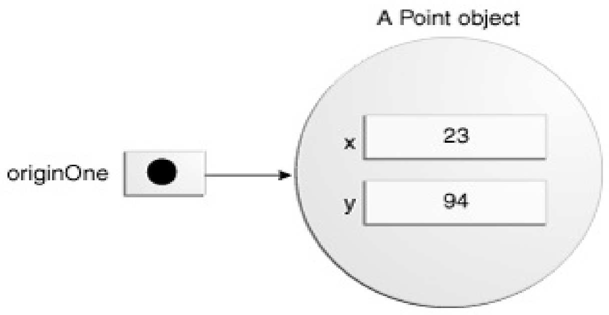
Here's the code for the `Point` class:

```
public class Point {
    public int x = 0;
    public int y = 0;
    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

This class contains a single constructor. You can recognize a constructor because its declaration uses the same name as the class and it has no return type. The constructor in the `Point` class takes two integer arguments, as declared by the code `(int a, int b)`. The following statement provides 23 and 94 as values for those arguments:

```
Point originOne = new Point(23, 94);
```

The result of executing this statement can be illustrated in the next figure:



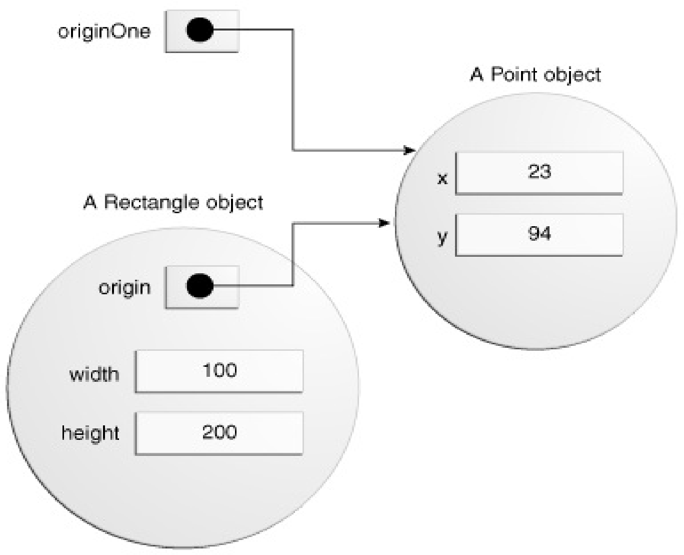
Here's the code for the `Rectangle` class, which contains four constructors:

```
public class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;

    // four constructors
    public Rectangle() {
        origin = new Point(0, 0);
    }
    public Rectangle(Point p) {
        origin = p;
    }
    public Rectangle(int w, int h) {
        origin = new Point(0, 0);
        width = w;
        height = h;
    }
    public Rectangle(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }

    // a method for moving the rectangle
    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }

    // a method for computing the area of the rectangle
    public int getArea() {
        return width * height;
    }
}
```

Each constructor lets you provide initial values for the rectangle's width and height, using both primitive and reference types. If a class has multiple constructors, they must have different signatures. The Java compiler differentiates the constructors based on the number and the type of the arguments. When the Java compiler encounters the following code, it knows to call the constructor in the `Rectangle` class that requires a `Point` argument followed by two integer arguments:

```
Rectangle rectOne = new Rectangle(originOne, 100, 200);
```

This calls one of `Rectangle`'s constructors that initializes `origin` to `originOne`. Also, the constructor sets `width` to 100 and `height` to 200. Now there are two references to the same `Point object`—an object can have multiple references to it, as shown in the next figure:



The following line of code calls the `Rectangle` constructor that requires two integer arguments, which provide the initial values for `width` and `height`. If you inspect the code within the constructor, you will see that it creates a new `Point` object whose `x` and `y` values are initialized to 0:

```
Rectangle rectTwo = new Rectangle(50, 100);
```

The `Rectangle` constructor used in the following statement doesn't take any arguments, so it's called a *no-argument constructor*:

```
Rectangle rect = new Rectangle();
```

All classes have at least one constructor. If a class does not explicitly declare any, the Java compiler automatically provides a no-argument constructor, called the *default constructor*. This default constructor calls the class parent's no-argument constructor, or the `Object` constructor if the class has no other parent. If the parent has no constructor (`Object` does have one), the compiler will reject the program.

Using Objects

Once you've created an object, you probably want to use it for something. You may need to use the value of one of its fields, change one of its fields, or call one of its methods to perform an action.

Referencing an Object's Fields

Object fields are accessed by their name. You must use a name that is unambiguous.

You may use a simple name for a field within its own class. For example, we can add a statement *within* the `Rectangle` class that prints the `width` and `height`:

```
System.out.println("Width and height are: " + width + ", " + height);
```

In this case, `width` and `height` are simple names.

Code that is outside the object's class must use an object reference or expression, followed by the dot (.) operator, followed by a simple field name, as in:

```
objectReference.fieldName
```

For example, the code in the `CreateObjectDemo` class is outside the code for the `Rectangle` class. So to refer to the `origin`, `width`, and `height` fields within the `Rectangle` object named `rectOne`, the `CreateObjectDemo` class must use the names `rectOne.origin`, `rectOne.width`, and `rectOne.height`, respectively. The program uses two of these names to display the `width` and the `height` of `rectOne`:

```
System.out.println("Width of rectOne: "  + rectOne.width);
System.out.println("Height of rectOne: " + rectOne.height);
```

Attempting to use the simple names `width` and `height` from the code in the `CreateObjectDemo` class doesn't make sense — those fields exist only within an object — and results in a compiler error.

Later, the program uses similar code to display information about `rectTwo`. Objects of the same type have their own copy of the same instance fields. Thus, each `Rectangle` object has fields named `origin`, `width`, and `height`. When you access an instance field through an object reference, you reference that particular object's field. The two objects `rectOne` and `rectTwo` in the `CreateObjectDemo` program have different `origin`, `width`, and `height` fields.

To access a field, you can use a named reference to an object, as in the previous examples, or you can use any expression that returns an object reference. Recall that the `new` operator returns a reference to an object. So you could use the value returned from new to access a new object's fields:

```
int height = new Rectangle().height;
```

This statement creates a new `Rectangle` object and immediately gets its height. In essence, the statement calculates the default height of a `Rectangle`. Note that after this statement has been executed, the program no longer has a reference to the created `Rectangle`, because the program never stored the reference anywhere. The object is unreferenced, and its resources are free to be recycled by the Java Virtual Machine.

Calling an Object's Methods

You also use an object reference to invoke an object's method. You append the method's simple name to the object reference, with an intervening dot operator (.). Also, you provide, within enclosing parentheses, any arguments to the method. If the method does not require any arguments, use empty parentheses.

```
objectReference.methodName(argumentList);
```

or:

```
objectReference.methodName();
```

The `Rectangle` class has two methods: `getArea()` to compute the rectangle's area and `move()` to change the rectangle's origin. Here's the `CreateObjectDemo` code that invokes these two methods:

```
System.out.println("Area of rectOne: " + rectOne.getArea());
...
rectTwo.move(40, 72);
```

The first statement invokes `rectOne`'s `getArea()` method and displays the results. The second line moves `rectTwo` because the `move()` method assigns new values to the object's `origin.x` and `origin.y`.

As with instance fields, *objectReference* must be a reference to an object. You can use a variable name, but you also can use any expression that returns an object reference. The `new` operator returns an object reference, so you can use the value returned from new to invoke a new object's methods:

```
new Rectangle(100, 50).getArea()
```

The expression `new Rectangle(100, 50)` returns an object reference that refers to a `Rectangle` object. As shown, you can use the dot notation to invoke the new `Rectangle`'s `getArea()` method to compute the area of the new rectangle.

Some methods, such as `getArea()`, return a value. For methods that return a value, you can use the method invocation in expressions. You can assign the return value to a variable, use it to make decisions, or control a loop. This code assigns the value returned by `getArea()` to the variable `areaOfRectangle`:

```
int areaOfRectangle = new Rectangle(100, 50).getArea();
```

Remember, invoking a method on a particular object is the same as sending a message to that object. In this case, the object that `getArea()` is invoked on is the rectangle returned by the constructor.

The Garbage Collector

Some object-oriented languages require that you keep track of all the objects you create and that you explicitly destroy them when they are no longer needed. Managing memory explicitly is tedious and error-prone. The Java platform allows you to create as many objects as you want (limited, of course, by what your system can handle), and you don't have to worry about destroying them. The Java runtime environment deletes objects when it determines that they are no longer being used. This process is called *garbage collection*.

An object is eligible for garbage collection when there are no more references to that object. References that are held in a variable are usually dropped when the variable goes out of scope. Or, you can explicitly drop an object reference by setting the variable to the special value `null`. Remember that a program can have multiple references to the same object; all references to an object must be dropped before the object is eligible for garbage collection.

The Java runtime environment has a garbage collector that periodically frees the memory used by objects that are no longer referenced. The garbage collector does its job automatically when it determines that the time is right.

More on Classes

This section covers more aspects of classes that depend on using object references and the `dot` operator that you learned about in the preceding sections on objects:

- Returning values from methods.
- The `this` keyword.
- Class vs. instance members.
- Access control.

Returning a Value from a Method

A method returns to the code that invoked it when it

- completes all the statements in the method,
- reaches a `return` statement, or
- throws an exception (covered later),

whichever occurs first.

You declare a method's return type in its method declaration. Within the body of the method, you use the `return` statement to return the value.

Any method declared `void` doesn't return a value. It does not need to contain a `return` statement, but it may do so. In such a case, a `return` statement can be used to branch out of a control flow block and exit the method and is simply used like this:

```
return;
```

If you try to return a value from a method that is declared `void`, you will get a compiler error.

Any method that is not declared `void` must contain a `return` statement with a corresponding return value, like this:

```
return returnValue;
```

The data type of the return value must match the method's declared return type; you can't return an integer value from a method declared to return a boolean.

The `getArea()` method in the `Rectangle Rectangle` class that was discussed in the sections on objects returns an integer:

```
    // a method for computing the area of the rectangle
    public int getArea() {
        return width * height;
    }
```

This method returns the integer that the expression `width*height` evaluates to.

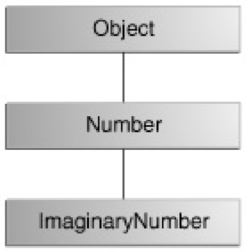
The `getArea` method returns a primitive type. A method can also return a reference type. For example, in a program to manipulate `Bicycle` objects, we might have a method like this:

```
public Bicycle seeWhosFastest(Bicycle myBike, Bicycle yourBike,
                              Environment env) {
    Bicycle fastest;
    // code to calculate which bike is
    // faster, given each bike's gear
    // and cadence and given the
    // environment (terrain and wind)
    return fastest;
}
```

Returning a Class or Interface

If this section confuses you, skip it and return to it after you have finished the lesson on interfaces and inheritance.

When a method uses a class name as its return type, such as `whosFastest` does, the class of the type of the returned object must be either a subclass of, or the exact class of, the return type. Suppose that you have a class hierarchy in which `ImaginaryNumber` is a subclass of `java.lang.Number`, which is in turn a subclass of `Object`, as illustrated in the following figure.



The class hierarchy for ImaginaryNumber

Now suppose that you have a method declared to return a `Number`:

```
public Number returnANumber() {
    ...
}
```

The `returnANumber` method can return an `ImaginaryNumber` but not an `Object`. `ImaginaryNumber` is a `Number` because it's a subclass of `Number`. However, an `Object` is not necessarily a `Number` — it could be a `String` or another type.

You can override a method and define it to return a subclass of the original method, like this:

```
public ImaginaryNumber returnANumber() {
    ...
}
```

This technique, called *covariant return type*, means that the return type is allowed to vary in the same direction as the subclass.

Note: You also can use interface names as return types. In this case, the object returned must implement the specified interface.

Using the this Keyword

Within an instance method or a constructor, `this` is a reference to the *current object* — the object whose method or constructor is being called. You can refer to any member of the current object from within an instance method or a constructor by using `this`.

Using `this` with a Field

The most common reason for using the `this` keyword is because a field is shadowed by a method or constructor parameter.

For example, the `Point` class was written like this

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

but it could have been written like this:

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Each argument to the constructor shadows one of the object's fields — inside the constructor **`x`** is a local copy of the constructor's first argument. To refer to the `Point` field **`x`**, the constructor must use **`this.x`**.

Using `this` with a Constructor

From within a constructor, you can also use the `this` keyword to call another constructor in the same class. Doing so is called an *explicit constructor invocation*. Here's another `Rectangle` class, with a different implementation from the one in the Objects section.

```
public class Rectangle {
    private int x, y;
    private int width, height;
    public Rectangle() {
        this(0, 0, 1, 1);
    }
    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
}
```

This class contains a set of constructors. Each constructor initializes some or all of the rectangle's member variables. The constructors provide a default value for any member variable whose initial value is not provided by an argument. For example, the no-argument constructor creates a 1x1 `Rectangle` at coordinates 0,0. The two-argument constructor calls the four-argument constructor, passing in the width and height but always using the 0,0 coordinates. As before, the compiler determines which constructor to call, based on the number and the type of arguments.

If present, the invocation of another constructor must be the first line in the constructor.

Controlling Access to Members of a Class

Access level modifiers determine whether other classes can use a particular field or invoke a particular method. There are two levels of access control:

- At the top level—`public`, or *package-private* (no explicit modifier).
- At the member level—`public`, `private`, `protected`, or *package-private* (no explicit modifier).

A class may be declared with the modifier `public`, in which case that class is visible to all classes everywhere. If a class has no modifier (the default, also known as *package-private*), it is visible only within its own package (packages are named groups of related classes — you will learn about them in a later lesson.)

At the member level, you can also use the `public` modifier or no modifier (*package-private*) just as with top-level classes, and with the same meaning. For members, there are two additional access modifiers: `private` and `protected`. The `private` modifier specifies that the member can only be accessed in its own class. The `protected` modifier specifies that the member can only be accessed within its own package (as with *package-private*) and, in addition, by a subclass of its class in another package.

The following table shows the access to members permitted by each modifier.

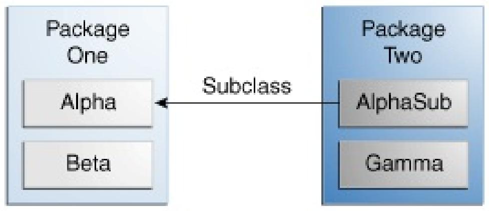
Access Levels

Modifier	**Class**	**Package**	**Subclass**	**World**
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

The first data column indicates whether the class itself has access to the member defined by the access level. As you can see, a class always has access to its own members. The second column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member. The third column indicates whether subclasses of the class declared outside this package have access to the member. The fourth column indicates whether all classes have access to the member.

Access levels affect you in two ways. First, when you use classes that come from another source, such as the classes in the Java platform, access levels determine which members of those classes your own classes can use. Second, when you write a class, you need to decide what access level every member variable and every method in your class should have.

Let's look at a collection of classes and see how access levels affect visibility. The following figure shows the four classes in this example and how they are related.



Classes and Packages of the Example Used to Illustrate Access Levels

The following table shows where the members of the Alpha class are visible for each of the access modifiers that can be applied to them.

Visibility

Modifier	**Alpha**	**Beta**	**Alphasub**	**Gamma**
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Tips on Choosing an Access Level:

If other programmers use your class, you want to ensure that errors from misuse cannot happen. Access levels can help you do this.

- Use the most restrictive access level that makes sense for a particular member. Use `private` unless you have a good reason not to.
- Avoid `public` fields except for constants. (Many of the examples in the tutorial use public fields. This may help to illustrate some points concisely, but is not recommended for production code.) Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.

Understanding Class Members

In this section, we discuss the use of the `static` keyword to create fields and methods that belong to the class, rather than to an instance of the class.

Class Variables

When a number of objects are created from the same class blueprint, they each have their own distinct copies of *instance variables*. In the case of the `Bicycle` class, the instance variables are `cadence`, `gear`, and `speed`. Each `Bicycle` object has its own values for these variables, stored in different memory locations.

Sometimes, you want to have variables that are common to all objects. This is accomplished with the `static` modifier. Fields that have the `static` modifier in their declaration are called *static fields* or *class variables*. They are associated with the class, rather than with any object. Every instance of the class shares a class variable, which is in one fixed location in memory. Any object can change the value of a class variable, but class variables can also be manipulated without creating an instance of the class.

For example, suppose you want to create a number of `Bicycle` objects and assign each a serial number, beginning with 1 for the first object. This ID number is unique to each object and is therefore an instance variable. At the same time, you need a field to keep track of how many `Bicycle` objects have been created so that you know what ID to assign to the next one. Such a field is not related to any individual object, but to the class as a whole. For this you need a class variable, `numberOfBicycles`, as follows:

```
public class Bicycle {

    private int cadence;
    private int gear;
    private int speed;

    // add an instance variable for the object ID
    private int id;

    // add a class variable for the
    // number of Bicycle objects instantiated
    private static int numberOfBicycles = 0;
        ...
}
```

Class variables are referenced by the class name itself, as in

```
Bicycle.numberOfBicycles
```

This makes it clear that they are class variables.

Note: You can also refer to static fields with an object reference like

```
myBike.numberOfBicycles
```

but this is discouraged because it does not make it clear that they are class variables.

You can use the `Bicycle` constructor to set the `id` instance variable and increment the `numberOfBicycles` class variable:

```
public class Bicycle {

    private int cadence;
    private int gear;
    private int speed;
    private int id;
    private static int numberOfBicycles = 0;

    public Bicycle(int startCadence, int startSpeed, int startGear){
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;

        // increment number of Bicycles
        // and assign ID number
        id = ++numberOfBicycles;
    }

    // new method to return the ID instance variable
    public int getID() {
        return id;
    }
        ...
}
```

Class Methods

The Java programming language supports static methods as well as static variables. Static methods, which have the `static` modifier in their declarations, should be invoked with the class name, without the need for creating an instance of the class, as in

```
ClassName.methodName(args)
```

Note: You can also refer to static methods with an object reference like

```
instanceName.methodName(args)
```

but this is discouraged because it does not make it clear that they are class methods.

A common use for static methods is to access static fields. For example, we could add a static method to the `Bicycle` class to access the `numberOfBicycles` static field:

```
public static int getNumberOfBicycles() {
    return numberOfBicycles;
}
```

Not all combinations of instance and class variables and methods are allowed:

- Instance methods can access instance variables and instance methods directly.
- Instance methods can access class variables and class methods directly.
- Class methods can access class variables and class methods directly.
- Class methods ***cannot*** access instance variables or instance methods directly—they must use an object reference. Also, class methods cannot use the `this` keyword as there is no instance for `this` to refer to.

Constants

The `static` modifier, in combination with the `final` modifier, is also used to define constants. The `final` modifier indicates that the value of this field cannot change.

For example, the following variable declaration defines a constant named `PI`, whose value is an approximation of pi (the ratio of the circumference of a circle to its diameter):

```
static final double PI = 3.141592653589793;
```

Constants defined in this way cannot be reassigned, and it is a compile-time error if your program tries to do so. By convention, the names of constant values are spelled in uppercase letters. If the name is composed of more than one word, the words are separated by an underscore (_).

Note: If a primitive type or a string is defined as a constant and the value is known at compile time, the compiler replaces the constant name everywhere in the code with its value. This is called a *compile-time constant*. If the value of the constant in the outside world changes (for example, if it is legislated that pi actually should be 3.975), you will need to recompile any classes that use this constant to get the current value.

The `Bicycle` Class

After all the modifications made in this section, the `Bicycle` class is now:

```
public class Bicycle {

    private int cadence;
    private int gear;
    private int speed;

    private int id;

    private static int numberOfBicycles = 0;


    public Bicycle(int startCadence,
                   int startSpeed,
                   int startGear){
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;

        id = ++numberOfBicycles;
    }

    public int getID() {
        return id;
    }

    public static int getNumberOfBicycles() {
        return numberOfBicycles;
    }

    public int getCadence(){
        return cadence;
    }

    public void setCadence(int newValue){
        cadence = newValue;
    }

    public int getGear(){
    return gear;
    }

    public void setGear(int newValue){
        gear = newValue;
    }

    public int getSpeed(){
        return speed;
    }

    public void applyBrake(int decrement){
        speed -= decrement;
    }

    public void speedUp(int increment){
        speed += increment;
    }
}
```

Initializing Fields

As you have seen, you can often provide an initial value for a field in its declaration:

```
public class BedAndBreakfast {

    // initialize to 10
    public static int capacity = 10;

    // initialize to false
    private boolean full = false;
}
```

This works well when the initialization value is available and the initialization can be put on one line. However, this form of initialization has limitations because of its simplicity. If initialization requires some logic (for example, error handling or a `for` loop to fill a complex array), simple assignment is inadequate. Instance variables can be initialized in constructors, where error handling or other logic can be used. To provide the same capability for class variables, the Java programming language includes *static initialization blocks*.

Note: It is not necessary to declare fields at the beginning of the class definition, although this is the most common practice. It is only necessary that they be declared and initialized before they are used.

Static Initialization Blocks

A *static initialization block* is a normal block of code enclosed in braces, `{ }`, and preceded by the `static` keyword. Here is an example:

```
static {
    // whatever code is needed for initialization goes here
}
```

A class can have any number of static initialization blocks, and they can appear anywhere in the class body. The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code.

There is an alternative to static blocks — you can write a private static method:

```
class Whatever {
    public static varType myVar = initializeClassVariable();

    private static varType initializeClassVariable() {

        // initialization code goes here
    }
}
```

The advantage of private static methods is that they can be reused later if you need to reinitialize the class variable.

Initializing Instance Members

Normally, you would put code to initialize an instance variable in a constructor. There are two alternatives to using a constructor to initialize instance variables: initializer blocks and final methods.

Initializer blocks for instance variables look just like static initializer blocks, but without the `static` keyword:

```
{
    // whatever code is needed for initialization goes here
}
```

The Java compiler copies initializer blocks into every constructor. Therefore, this approach can be used to share a block of code between multiple constructors.

A *final method* cannot be overridden in a subclass. This is discussed in the lesson on interfaces and inheritance. Here is an example of using a final method for initializing an instance variable:

```
class Whatever {
    private varType myVar = initializeInstanceVariable();

    protected final varType initializeInstanceVariable() {

        // initialization code goes here
    }
}
```

This is especially useful if subclasses might want to reuse the initialization method. The method is final because calling non-final methods during instance initialization can cause problems.

Summary of Creating and Using Classes and Objects

A class declaration names the class and encloses the class body between braces. The class name can be preceded by modifiers. The class body contains fields, methods, and constructors for the class. A class uses fields to contain state information and uses methods to implement behavior. Constructors that initialize a new instance of a class use the name of the class and look like methods without a return type.

You control access to classes and members in the same way: by using an access modifier such as `public` in their declaration.

You specify a class variable or a class method by using the `static` keyword in the member's declaration. A member that is not declared as `static` is implicitly an instance member. Class variables are shared by all instances of a class and can be accessed through the class name as well as an instance reference. Instances of a class get their own copy of each instance variable, which must be accessed through an instance reference.

You create an object from a class by using the `new` operator and a constructor. The new operator returns a reference to the object that was created. You can assign the reference to a variable or use it directly.

Instance variables and methods that are accessible to code outside of the class that they are declared in can be referred to by using a qualified name. The qualified name of an instance variable looks like this:

```
objectReference.variableName
```

The qualified name of a method looks like this:

```
objectReference.methodName(argumentList)
```

or:

```
objectReference.methodName()
```

The garbage collector automatically cleans up unused objects. An object is unused if the program holds no more references to it. You can explicitly drop a reference by setting the variable holding the reference to `null`.

Questions and Exercises: Classes

Questions

1. Consider the following class:

```
public class IdentifyMyParts {
    public static int x = 7;
    public int y = 3;
}
```

a. What are the class variables?
b. What are the instance variables?
c. What is the output from the following code:

```
IdentifyMyParts a = new IdentifyMyParts();
IdentifyMyParts b = new IdentifyMyParts();
a.y = 5;
b.y = 6;
a.x = 1;
b.x = 2;
System.out.println("a.y = " + a.y);
System.out.println("b.y = " + b.y);
System.out.println("a.x = " + a.x);
System.out.println("b.x = " + b.x);
System.out.println("IdentifyMyParts.x = " + IdentifyMyParts.x);
```

Exercises

1. Write a class whose instances represent a single playing card from a deck of cards. Playing cards have two distinguishing properties: rank and suit. Be sure to keep your solution as you will be asked to rewrite it in Enum Types.

Hint: You can use the `assert` statement to check your assignments. You write:

```
assert (boolean expression to test);
```

If the boolean expression is false, you will get an error message. For example,

```
assert toString(ACE) == "Ace";
```

should return `true`, so there will be no error message.
If you use the `assert` statement, you must run your program with the `ea` flag:

```
java -ea YourProgram.class
```

---

2. Write a class whose instances represent a **full** deck of cards. You should also keep this solution.
3. 3. Write a small program to test your deck and card classes. The program can be as simple as creating a deck of cards and displaying its cards.

Check your answers.

Questions

1. Consider the following class:

```
public class IdentifyMyParts {
    public static int x = 7;
    public int y = 3;
}
```

a. Question: What are the class variables?
Answer: x
b. Question: What are the instance variables?
Answer: y
c. Question: What is the output from the following code:

```
IdentifyMyParts a = new IdentifyMyParts();
IdentifyMyParts b = new IdentifyMyParts();
a.y = 5;
b.y = 6;
a.x = 1;
b.x = 2;
System.out.println("a.y = " + a.y);
System.out.println("b.y = " + b.y);
System.out.println("a.x = " + a.x);
System.out.println("b.x = " + b.x);
System.out.println("IdentifyMyParts.x = " + IdentifyMyParts.x);
```

Answer: Here is the output:

```
 a.y = 5
 b.y = 6
 a.x = 2
 b.x = 2
 IdentifyMyParts.x = 2
```

Because `x` is defined as a `public static int` in the class `IdentifyMyParts`, every reference to `x` will have the value that was last assigned because `x` is a static variable (and therefore a class variable) shared across all instances of the class. That is, there is only one `x`: when the value of `x` changes in any instance it affects the value of `x` for all instances of `IdentifyMyParts`.
This is covered in the Class Variables section of Understanding Instance and Class Members.

Exercises

1. Exercise: Write a class whose instances represent a single playing card from a deck of cards. Playing cards have two distinguishing properties: rank and suit. Be sure to keep your solution as you will be asked to rewrite it in Enum Types.
Answer: `Card.java`.
2. Exercise: Write a class whose instances represents a **full** deck of cards. You should also keep this solution.
Answer: See `Deck.java`.
3. Exercise: Write a small program to test your deck and card classes. The program can be as simple as creating a deck of cards and displaying its cards.
Answer: See `DisplayDeck.java`.

Questions and Exercises: Objects

Questions

1. What's wrong with the following program?

```
public class SomethingIsWrong {
    public static void main(String[] args) {
        Rectangle myRect;
        myRect.width = 40;
        myRect.height = 50;
        System.out.println("myRect's area is " + myRect.area());
    }
}
```

2. The following code creates one array and one string object. How many references to those objects exist after the code executes? Is either object eligible for garbage collection?

```
...
String[] students = new String[10];
String studentName = "Peter Parker";
students[0] = studentName;
studentName = null;
...
```

3. How does a program destroy an object that it creates?

Exercises

1. Fix the program called `SomethingIsWrong` shown in Question 1.

2. Given the following class, called `NumberHolder`, write some code that creates an instance of the class, initializes its two member variables, and then displays the value of each member variable.

```
public class NumberHolder {
    public int anInt;
    public float aFloat;
}
```

Check your answers.

Questions

1. Question: What's wrong with the following program?

```
public class SomethingIsWrong {
    public static void main(String[] args) {
        Rectangle myRect;
        myRect.width = 40;
        myRect.height = 50;
        System.out.println("myRect's area is " + myRect.area());
    }
}
```

Answer: The code never creates a `Rectangle` object. With this simple program, the compiler generates an error. However, in a more realistic situation, `myRect` might be initialized to `null` in one place, say in a constructor, and used later. In that case, the program will compile just fine, but will generate a `NullPointerException` at runtime.

2. Question: The following code creates one array and one string object. How many references to those objects exist after the code executes? Is either object eligible for garbage collection?

```
...
String[] students = new String[10];
String studentName = "Peter Smith";
students[0] = studentName;
studentName = null;
...
```

Answer: There is one reference to the `students` array and that array has one reference to the string `Peter Smith`. Neither object is eligible for garbage collection.

3. Question: How does a program destroy an object that it creates?

Answer: A program does not explicitly destroy objects. A program can set all references to an object to `null` so that it becomes eligible for garbage collection. But the program does not actually destroy objects.

Exercises

1. Exercise: Fix the program called `SomethingIsWrong` shown in Question 1.

Answer: See `SomethingIsRight`:

```
public class SomethingIsRight {
    public static void main(String[] args) {
        Rectangle myRect = new Rectangle();
        myRect.width = 40;
        myRect.height = 50;
        System.out.println("myRect's area is " + myRect.area());
    }
}
```

2. Exercise: Given the following class, called `NumberHolder`, write some code that creates an instance of the class, initializes its two member variables, and then displays the value of each member variable.

```
public class NumberHolder {
    public int anInt;
    public float aFloat;
}
```

Answer: See `NumberHolderDisplay`:

```
public class NumberHolderDisplay {
    public static void main(String[] args) {
        NumberHolder aNumberHolder = new NumberHolder();
        aNumberHolder.anInt = 1;
        aNumberHolder.aFloat = 2.3f;
        System.out.println(aNumberHolder.anInt);
        System.out.println(aNumberHolder.aFloat);
    }
}
```

Nested Classes

The Java programming language allows you to define a class within another class. Such a class is called a *nested class* and is illustrated here:

```
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

Terminology: Nested classes are divided into two categories: static and non-static. Nested classes that are declared `static` are called *static nested classes*. Non-static nested classes are called *inner classes*.

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
    class InnerClass {
        ...
    }
}
```

A nested class is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private. Static nested classes do not have access to other members of the enclosing class. As a member of the `OuterClass`, a nested class can be declared `private`, `public`, `protected`, or *package private*. (Recall that outer classes can only be declared `public` or *package private*.)

Why Use Nested Classes?

Compelling reasons for using nested classes include the following:

- **It is a way of logically grouping classes that are only used in one place**: If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
- **It increases encapsulation**: Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared `private`. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
- **It can lead to more readable and maintainable code**: Nesting small classes within top-level classes places the code closer to where it is used.

Static Nested Classes

As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.

Note:A static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class. In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.

Static nested classes are accessed using the enclosing class name:

```
OuterClass.StaticNestedClass
```

For example, to create an object for the static nested class, use this syntax:

```
OuterClass.StaticNestedClass nestedObject =
     new OuterClass.StaticNestedClass();
```

Inner Classes

As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields. Also, because an inner class is associated with an instance, it cannot define any static members itself.

Objects that are instances of an inner class exist *within* an instance of the outer class. Consider the following classes:

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

An instance of `InnerClass` can exist only within an instance of `OuterClass` and has direct access to the methods and fields of its enclosing instance.

To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

There are two special kinds of inner classes: local classes and anonymous classes.

Shadowing

If a declaration of a type (such as a member variable or a parameter name) in a particular scope (such as an inner class or a method definition) has the same name as another declaration in the enclosing scope, then the declaration *shadows* the declaration of the enclosing scope. You cannot refer to a shadowed declaration by its name alone. The following example, `ShadowTest`, demonstrates this:

```
public class ShadowTest {

    public int x = 0;

    class FirstLevel {

        public int x = 1;

        void methodInFirstLevel(int x) {
            System.out.println("x = " + x);
            System.out.println("this.x = " + this.x);
            System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);
        }
    }

    public static void main(String... args) {
        ShadowTest st = new ShadowTest();
        ShadowTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
```

The following is the output of this example:

```
x = 23
this.x = 1
ShadowTest.this.x = 0
```

This example defines three variables named `x`: the member variable of the class `ShadowTest`, the member variable of the inner class `FirstLevel`, and the parameter in the method `methodInFirstLevel`. The variable `x` defined as a parameter of the method `methodInFirstLevel` shadows the variable of the inner class `FirstLevel`. Consequently, when you use the variable `x` in the method `methodInFirstLevel`, it refers to the method parameter. To refer to the member variable of the inner class `FirstLevel`, use the keyword `this` to represent the enclosing scope:

```
System.out.println("this.x = " + this.x);
```

Refer to member variables that enclose larger scopes by the class name to which they belong. For example, the following statement accesses the member variable of the class `ShadowTest` from the method `methodInFirstLevel`:

```
System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);
```

Serialization

Serialization of inner classes, including local and anonymous classes, is strongly discouraged. When the Java compiler compiles certain constructs, such as inner classes, it creates *synthetic constructs*; these are classes, methods, fields, and other constructs that do not have a corresponding construct in the source code. Synthetic constructs enable Java compilers to implement new Java language features without changes to the JVM. However, synthetic constructs can vary among different Java compiler implementations, which means that `.class` files can vary among different implementations as well. Consequently, you may have compatibility issues if you serialize an inner class and then deserialize it with a different JRE implementation. See the section Implicit and Synthetic Parameters in the section Obtaining Names of Method Parameters for more information about the synthetic constructs generated when an inner class is compiled.

Inner Class Example

To see an inner class in use, first consider an array. In the following example, you create an array, fill it with integer values, and then output only values of even indices of the array in ascending order.

The `DataStructure.java` example that follows consists of:

- The `DataStructure` outer class, which includes a constructor to create an instance of `DataStructure` containing an array filled with consecutive integer values (0, 1, 2, 3, and so on) and a method that prints elements of the array that have an even index value.
- The `EvenIterator` inner class, which implements the `DataStructureIterator` interface, which extends the `Iterator`< `Integer`> interface. Iterators are used to step through a data structure and typically have methods to test for the last element, retrieve the current element, and move to the next element.
- A `main` method that instantiates a `DataStructure` object (`ds`), then invokes the `printEven` method to print elements of the array `arrayOfInts` that have an even index value.

```
public class DataStructure {

    // Create an array
    private final static int SIZE = 15;
    private int[] arrayOfInts = new int[SIZE];

    public DataStructure() {
        // fill the array with ascending integer values
        for (int i = 0; i < SIZE; i++) {
            arrayOfInts[i] = i;
        }
    }

    public void printEven() {

        // Print out values of even indices of the array
        DataStructureIterator iterator = this.new EvenIterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
        System.out.println();
    }

    interface DataStructureIterator extends java.util.Iterator<Integer> { }

    // Inner class implements the DataStructureIterator interface,
    // which extends the Iterator<Integer> interface

    private class EvenIterator implements DataStructureIterator {

        // Start stepping through the array from the beginning
        private int nextIndex = 0;

        public boolean hasNext() {

            // Check if the current element is the last in the array
            return (nextIndex <= SIZE - 1);
        }

        public Integer next() {

            // Record a value of an even index of the array
            Integer retValue = Integer.valueOf(arrayOfInts[nextIndex]);

            // Get the next even element
            nextIndex += 2;
            return retValue;
        }
    }

    public static void main(String s[]) {

        // Fill the array with integer values and print out only
        // values of even indices
        DataStructure ds = new DataStructure();
        ds.printEven();
    }
}
```

The output is:

```
0 2 4 6 8 10 12 14
```

Note that the `EvenIterator` class refers directly to the `arrayOfInts` instance variable of the `DataStructure` object.

You can use inner classes to implement helper classes such as the one shown in the this example. To handle user interface events, you must know how to use inner classes, because the event-handling mechanism makes extensive use of them.

Local and Anonymous Classes

There are two additional types of inner classes. You can declare an inner class within the body of a method. These classes are known as local classes. You can also declare an inner class within the body of a method without naming the class. These classes are known as anonymous classes.

Modifiers

You can use the same modifiers for inner classes that you use for other members of the outer class. For example, you can use the access specifiers `private`, `public`, and `protected` to restrict access to inner classes, just as you use them to restrict access do to other class members.

Local Classes

Local classes are classes that are defined in a *block*, which is a group of zero or more statements between balanced braces. You typically find local classes defined in the body of a method.

This section covers the following topics:

- Declaring Local Classes
- Accessing Members of an Enclosing Class
  - Shadowing and Local Classes
- Local Classes Are Similar To Inner Classes

Declaring Local Classes

You can define a local class inside any block (see Expressions, Statements, and Blocks for more information). For example, you can define a local class in a method body, a `for` loop, or an `if` clause.

The following example, `LocalClassExample`, validates two phone numbers. It defines the local class `PhoneNumber` in the method `validatePhoneNumber`:

```
public class LocalClassExample {

    static String regularExpression = "[^0-9]";

    public static void validatePhoneNumber(
        String phoneNumber1, String phoneNumber2) {

        final int numberLength = 10;

        // Valid in JDK 8 and later:

        // int numberLength = 10;

        class PhoneNumber {

            String formattedPhoneNumber = null;

            PhoneNumber(String phoneNumber){
                // numberLength = 7;
                String currentNumber = phoneNumber.replaceAll(
                  regularExpression, "");
                if (currentNumber.length() == numberLength)
                    formattedPhoneNumber = currentNumber;
                else
                    formattedPhoneNumber = null;
            }

            public String getNumber() {
                return formattedPhoneNumber;
            }

            // Valid in JDK 8 and later:

//            public void printOriginalNumbers() {
//                System.out.println("Original nubmers are " + phoneNumber1 +
//                    " and " + phoneNumber2);
//            }
```

```
        }

        PhoneNumber myNumber1 = new PhoneNumber(phoneNumber1);
        PhoneNumber myNumber2 = new PhoneNumber(phoneNumber2);

        // Valid in JDK 8 and later:

//        myNumber1.printOriginalNumbers();

        if (myNumber1.getNumber() == null)
            System.out.println("First number is invalid");
        else
            System.out.println("First number is " + myNumber1.getNumber());
        if (myNumber2.getNumber() == null)
            System.out.println("Second number is invalid");
        else
            System.out.println("Second number is " + myNumber2.getNumber());

    }

    public static void main(String... args) {
        validatePhoneNumber("123-456-7890", "456-7890");
    }
}
```

The example validates a phone number by first removing all characters from the phone number except the digits 0 through 9. After, it checks whether the phone number contains exactly ten digits (the length of a phone number in North America). This example prints the following:

```
First number is 1234567890
Second number is invalid
```

Accessing Members of an Enclosing Class

A local class has access to the members of its enclosing class. In the previous example, the `PhoneNumber` constructor accesses the member `LocalClassExample.regularExpression`.

In addition, a local class has access to local variables. However, a local class can only access local variables that are declared final. When a local class accesses a local variable or parameter of the enclosing block, it *captures* that variable or parameter. For example, the `PhoneNumber` constructor can access the local variable `numberLength` because it is declared final; `numberLength` is a *captured variable*.

However, starting in Java SE 8, a local class can access local variables and parameters of the enclosing block that are final or *effectively final*. A variable or parameter whose value is never changed after it is initialized is effectively final. For example, suppose that the variable `numberLength` is not declared final, and you add the highlighted assignment statement in the `PhoneNumber` constructor:

```
PhoneNumber(String phoneNumber) {
    numberLength = 7;
    String currentNumber = phoneNumber.replaceAll(
        regularExpression, "");
    if (currentNumber.length() == numberLength)
        formattedPhoneNumber = currentNumber;
    else
        formattedPhoneNumber = null;
}
```

Because of this assignment statement, the variable `numberLength` is not effectively final anymore. As a result, the Java compiler generates an error message similar to "local variables referenced from an inner class must be final or effectively final" where the inner class `PhoneNumber` tries to access the `numberLength` variable:

```
if (currentNumber.length() == numberLength)
```

Starting in Java SE 8, if you declare the local class in a method, it can access the method's parameters. For example, you can define the following method in the `PhoneNumber` local class:

```
public void printOriginalNumbers() {
    System.out.println("Original numbers are " + phoneNumber1 +
        " and " + phoneNumber2);
}
```

The method `printOriginalNumbers` accesses the parameters `phoneNumber1` and `phoneNumber2` of the method `validatePhoneNumber`.

Shadowing and Local Classes

Declarations of a type (such as a variable) in a local class shadow declarations in the enclosing scope that have the same name. See Shadowing for more information.

Local Classes Are Similar To Inner Classes

Local classes are similar to inner classes because they cannot define or declare any static members. Local classes in static methods, such as the class `PhoneNumber`, which is defined in the static method `validatePhoneNumber`, can only refer to static members of the enclosing class. For example, if you do not define the member variable `regularExpression` as static, then the Java compiler generates an error similar to "non-static variable `regularExpression` cannot be referenced from a static context."

Local classes are non-static because they have access to instance members of the enclosing block. Consequently, they cannot contain most kinds of static declarations.

You cannot declare an interface inside a block; interfaces are inherently static. For example, the following code excerpt does not compile because the interface `HelloThere` is defined inside the body of the method `greetInEnglish`:

```
    public void greetInEnglish() {
        interface HelloThere {
           public void greet();
        }
        class EnglishHelloThere implements HelloThere {
            public void greet() {
                System.out.println("Hello " + name);
            }
        }
        HelloThere myGreeting = new EnglishHelloThere();
        myGreeting.greet();
    }
```

You cannot declare static initializers or member interfaces in a local class. The following code excerpt does not compile because the method `EnglishGoodbye.sayGoodbye` is declared `static`. The compiler generates an error similar to "modifier 'static' is only allowed in constant variable declaration" when it encounters this method definition:

```
    public void sayGoodbyeInEnglish() {
        class EnglishGoodbye {
            public static void sayGoodbye() {
                System.out.println("Bye bye");
            }
        }
        EnglishGoodbye.sayGoodbye();
    }
```

A local class can have static members provided that they are constant variables. (A *constant variable* is a variable of primitive type or type `String` that is declared final and initialized with a compile-time constant expression. A compile-time constant expression is typically a string or an arithmetic expression that can be evaluated at compile time. See Understanding Class Members for more information.) The following code excerpt compiles because the static member `EnglishGoodbye.farewell` is a constant variable:

```
    public void sayGoodbyeInEnglish() {
        class EnglishGoodbye {
            public static final String farewell = "Bye bye";
            public void sayGoodbye() {
                System.out.println(farewell);
            }
        }
        EnglishGoodbye myEnglishGoodbye = new EnglishGoodbye();
        myEnglishGoodbye.sayGoodbye();
    }
```

Anonymous Classes

Anonymous classes enable you to make your code more concise. They enable you to declare and instantiate a class at the same time. They are like local classes except that they do not have a name. Use them if you need to use a local class only once.

This section covers the following topics:

- Declaring Anonymous Classes
- Syntax of Anonymous Classes
- Accessing Local Variables of the Enclosing Scope, and Declaring and Accessing Members of the Anonymous Class
- Examples of Anonymous Classes

Declaring Anonymous Classes

While local classes are class declarations, anonymous classes are expressions, which means that you define the class in another expression. The following example, `HelloWorldAnonymousClasses`, uses anonymous classes in the initialization statements of the local variables `frenchGreeting` and `spanishGreeting`, but uses a local class for the initialization of the variable `englishGreeting`:

```
public class HelloWorldAnonymousClasses {

    interface HelloWorld {
        public void greet();
        public void greetSomeone(String someone);
    }

    public void sayHello() {

        class EnglishGreeting implements HelloWorld {
            String name = "world";
            public void greet() {
                greetSomeone("world");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Hello " + name);
            }
        }

        HelloWorld englishGreeting = new EnglishGreeting();

        HelloWorld frenchGreeting = new HelloWorld() {
            String name = "tout le monde";
            public void greet() {
                greetSomeone("tout le monde");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Salut " + name);
            }
        };

        HelloWorld spanishGreeting = new HelloWorld() {
            String name = "mundo";
            public void greet() {
                greetSomeone("mundo");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Hola, " + name);
            }
        };
        englishGreeting.greet();
        frenchGreeting.greetSomeone("Fred");
        spanishGreeting.greet();
    }

    public static void main(String... args) {
        HelloWorldAnonymousClasses myApp =
            new HelloWorldAnonymousClasses();
        myApp.sayHello();
    }
}
```

Syntax of Anonymous Classes

As mentioned previously, an anonymous class is an expression. The syntax of an anonymous class expression is like the invocation of a constructor, except that there is a class definition contained in a block of code.

Consider the instantiation of the `frenchGreeting` object:

```
        HelloWorld frenchGreeting = new HelloWorld() {
            String name = "tout le monde";
            public void greet() {
                greetSomeone("tout le monde");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Salut " + name);
            }
        };
```

The anonymous class expression consists of the following:

- The `new` operator
- The name of an interface to implement or a class to extend. In this example, the anonymous class is implementing the interface `HelloWorld`.
- Parentheses that contain the arguments to a constructor, just like a normal class instance creation expression. **Note**: When you implement an interface, there is no constructor, so you use an empty pair of parentheses, as in this example.
- A body, which is a class declaration body. More specifically, in the body, method declarations are allowed but statements are not.

Because an anonymous class definition is an expression, it must be part of a statement. In this example, the anonymous class expression is part of the statement that instantiates the `frenchGreeting` object. (This explains why there is a semicolon after the closing brace.)

Accessing Local Variables of the Enclosing Scope, and Declaring and Accessing Members of the Anonymous Class

Like local classes, anonymous classes can capture variables; they have the same access to local variables of the enclosing scope:

- An anonymous class has access to the members of its enclosing class.
- An anonymous class cannot access local variables in its enclosing scope that are not declared as `final` or effectively final.
- Like a nested class, a declaration of a type (such as a variable) in an anonymous class shadows any other declarations in the enclosing scope that have the same name. See Shadowing for more information.

Anonymous classes also have the same restrictions as local classes with respect to their members:

- You cannot declare static initializers or member interfaces in an anonymous class.
- An anonymous class can have static members provided that they are constant variables.

Note that you can declare the following in anonymous classes:

- Fields
- Extra methods (even if they do not implement any methods of the supertype)
- Instance initializers
- Local classes

However, you cannot declare constructors in an anonymous class.

Examples of Anonymous Classes

Anonymous classes are often used in graphical user interface (GUI) applications.

Consider the JavaFX example `HelloWorld.java` (from the section Hello World, JavaFX Style from Getting Started with JavaFX). This sample creates a frame that contains a **Say 'Hello World'** button. The anonymous class expression is highlighted:

```
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World!");
        Button btn = new Button();
        btn.setText("Say 'Hello World'");
        btn.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });

        StackPane root = new StackPane();
        root.getChildren().add(btn);
        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }
}
```

In this example, the method invocation `btn.setOnAction` specifies what happens when you select the **Say 'Hello World'** button. This method requires an object of type `EventHandler<ActionEvent>`. The `EventHandler<ActionEvent>` interface contains only one method, handle. Instead of implementing this method with a new class, the example uses an anonymous class expression. Notice that this expression is the argument passed to the `btn.setOnAction` method.

Because the `EventHandler<ActionEvent>` interface contains only one method, you can use a lambda expression instead of an anonymous class expression. See the section Lambda Expressions for more information.

Anonymous classes are ideal for implementing an interface that contains two or more methods. The following JavaFX example is from the section Customization of UI Controls. The highlighted code creates a text field that only accepts numeric values. It redefines the default implementation of the `TextField` class with an anonymous class by overriding the `replaceText` and `replaceSelection` methods inherited from the `TextInputControl` class.

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class CustomTextFieldSample extends Application {

    final static Label label = new Label();

    @Override
    public void start(Stage stage) {
        Group root = new Group();
        Scene scene = new Scene(root, 300, 150);
        stage.setScene(scene);
        stage.setTitle("Text Field Sample");

        GridPane grid = new GridPane();
        grid.setPadding(new Insets(10, 10, 10, 10));
        grid.setVgap(5);
        grid.setHgap(5);

        scene.setRoot(grid);
        final Label dollar = new Label("$");
        GridPane.setConstraints(dollar, 0, 0);
        grid.getChildren().add(dollar);

        final TextField sum = new TextField() {
            @Override
            public void replaceText(int start, int end, String text) {
                if (!text.matches("[a-z, A-Z]")) {
                    super.replaceText(start, end, text);
                }
                label.setText("Enter a numeric value");
            }

            @Override
            public void replaceSelection(String text) {
                if (!text.matches("[a-z, A-Z]")) {
                    super.replaceSelection(text);
                }
            }
        };

        sum.setPromptText("Enter the total");
        sum.setPrefColumnCount(10);
        GridPane.setConstraints(sum, 1, 0);
        grid.getChildren().add(sum);

        Button submit = new Button("Submit");
        GridPane.setConstraints(submit, 2, 0);
        grid.getChildren().add(submit);

        submit.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent e) {
                label.setText(null);
            }
        });

        GridPane.setConstraints(label, 0, 1);
        GridPane.setColumnSpan(label, 3);
        grid.getChildren().add(label);

        scene.setRoot(grid);
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Lambda Expressions

One issue with anonymous classes is that if the implementation of your anonymous class is very simple, such as an interface that contains only one method, then the syntax of anonymous classes may seem unwieldy and unclear. In these cases, you're usually trying to pass functionality as an argument to another method, such as what action should be taken when someone clicks a button. Lambda expressions enable you to do this, to treat functionality as method argument, or code as data.

The previous section, Anonymous Classes, shows you how to implement a base class without giving it a name. Although this is often more concise than a named class, for classes with only one method, even an anonymous class seems a bit excessive and cumbersome. Lambda expressions let you express instances of single-method classes more compactly.

This section covers the following topics:

- Ideal Use Case for Lambda Expressions
  - Approach 1: Create Methods That Search for Members That Match One Characteristic
  - Approach 2: Create More Generalized Search Methods
  - Approach 3: Specify Search Criteria Code in a Local Class
  - Approach 4: Specify Search Criteria Code in an Anonymous Class
  - Approach 5: Specify Search Criteria Code with a Lambda Expression
  - Approach 6: Use Standard Functional Interfaces with Lambda Expressions
  - Approach 7: Use Lambda Expressions Throughout Your Application
  - Approach 8: Use Generics More Extensively
  - Approach 9: Use Aggregate Operations That Accept Lambda Expressions as Parameters
- Lambda Expressions in GUI Applications
- Syntax of Lambda Expressions
- Accessing Local Variables of the Enclosing Scope
- Target Typing
  - Target Types and Method Arguments
- Serialization

Ideal Use Case for Lambda Expressions

Suppose that you are creating a social networking application. You want to create a feature that enables an administrator to perform any kind of action, such as sending a message, on members of the social networking application that satisfy certain criteria. The following table describes this use case in detail:

Field	Description
Name	Perform action on selected members
Primary Actor	Administrator
Preconditions	Administrator is logged in to the system.

Postconditions	Action is performed only on members that fit the specified criteria.
Main Success Scenario	**1.** Administrator specifies criteria of members on which to perform a certain action. **2.** Administrator specifies an action to perform on those selected members. **3.** Administrator selects the **Submit** button. **4.** The system finds all members that match the specified criteria. **5.** The system performs the specified action on all matching members.
Extensions	1a. Administrator has an option to preview those members who match the specified criteria before he or she specifies the action to be performed or before selecting the **Submit** button.
Frequency of Occurrence	Many times during the day.

Suppose that members of this social networking application are represented by the following `Person` class:

```
public class Person {

    public enum Sex {
        MALE, FEMALE
    }

    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    public int getAge() {
        // ...
    }

    public void printPerson() {
        // ...
    }
}
```

Suppose that the members of your social networking application are stored in a `List<Person>` instance.

This section begins with a naive approach to this use case. It improves upon this approach with local and anonymous classes, and then finishes with an efficient and concise approach using lambda expressions. Find the code excerpts described in this section in the example `RosterTest`.

Approach 1: Create Methods That Search for Members That Match One Characteristic

One simplistic approach is to create several methods; each method searches for members that match one characteristic, such as gender or age. The following method prints members that are older than a specified age:

```
public static void printPersonsOlderThan(List<Person> roster, int age) {
    for (Person p : roster) {
        if (p.getAge() >= age) {
            p.printPerson();
        }
    }
}
```

Note: A List is an ordered Collection. A *collection* is an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. For more information about collections, see the Collections trail.

This approach can potentially make your application *brittle*, which is the likelihood of an application not working because of the introduction of updates (such as newer data types). Suppose that you upgrade your application and change the structure of the `Person` class such that it contains different member variables; perhaps the class records and measures ages with a different data type or algorithm. You would have to rewrite a lot of your API to accommodate this change. In addition, this approach is unnecessarily restrictive; what if you wanted to print members younger than a certain age, for example?

Approach 2: Create More Generalized Search Methods

The following method is more generic than `printPersonsOlderThan`; it prints members within a specified range of ages:

```
public static void printPersonsWithinAgeRange(
    List<Person> roster, int low, int high) {
    for (Person p : roster) {
        if (low <= p.getAge() && p.getAge() < high) {
            p.printPerson();
        }
    }
}
```

What if you want to print members of a specified sex, or a combination of a specified gender and age range? What if you decide to change the `Person` class and add other attributes such as relationship status or geographical location? Although this method is more generic than `printPersonsOlderThan`, trying to create a separate method for each possible search query can still lead to brittle code. You can instead separate the code that specifies the criteria for which you want to search in a different class.

Approach 3: Specify Search Criteria Code in a Local Class

The following method prints members that match search criteria that you specify:

```
public static void printPersons(
    List<Person> roster, CheckPerson tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

This method checks each `Person` instance contained in the `List` parameter `roster` whether it satisfies the search criteria specified in the `CheckPerson` parameter `tester` by invoking the method tester.test. If the method tester.test returns a true value, then the method printPersons is invoked on the Person instance.

To specify the search criteria, you implement the CheckPerson interface:

```
interface CheckPerson {
    boolean test(Person p);
}
```

The following class implements the CheckPerson interface by specifying an implementation for the method test. This method filters members that are eligible for Selective Service in the United States: it returns a true value if its Person parameter is male and between the ages of 18 and 25:

```
class CheckPersonEligibleForSelectiveService implements CheckPerson {
    public boolean test(Person p) {
        return p.gender == Person.Sex.MALE &&
            p.getAge() >= 18 &&
            p.getAge() <= 25;
    }
}
```

To use this class, you create a new instance of it and invoke the printPersons method:

```
printPersons(
    roster, new CheckPersonEligibleForSelectiveService());
```

Although this approach is less brittleyou don't have to rewrite methods if you change the structure of the Personyou still have additional code: a new interface and a local class for each search you plan to perform in your application. Because CheckPersonEligibleForSelectiveService implements an interface, you can use an anonymous class instead of a local class and bypass the need to declare a new class for each search.

Approach 4: Specify Search Criteria Code in an Anonymous Class

One of the arguments of the following invocation of the method printPersons is an anonymous class that filters members that are eligible for Selective Service in the United States: those who are male and between the ages of 18 and 25:

```
printPersons(
    roster,
    new CheckPerson() {
        public boolean test(Person p) {
            return p.getGender() == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 25;
        }
    }
);
```

This approach reduces the amount of code required because you don't have to create a new class for each search that you want to perform. However, the syntax of anonymous classes is bulky considering that the CheckPerson interface contains only one method. In this case, you can use a lambda expression instead of an anonymous class, as described in the next section.

Approach 5: Specify Search Criteria Code with a Lambda Expression

The `CheckPerson` interface is a *functional interface*. A functional interface is any interface that contains only one abstract method. (A functional interface may contain one or more default methods or static methods.) Because a functional interface contains only one abstract method, you can omit the name of that method when you implement it. To do this, instead of using an anonymous class expression, you use a *lambda expression*, which is highlighted in the following method invocation:

```
printPersons(
    roster,
    (Person p) -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

See Syntax of Lambda Expressions for information about how to define lambda expressions.

You can use a standard functional interface in place of the interface `CheckPerson`, which reduces even further the amount of code required.

Approach 6: Use Standard Functional Interfaces with Lambda Expressions

Reconsider the `CheckPerson` interface:

```
interface CheckPerson {
    boolean test(Person p);
}
```

This is a very simple interface. It's a functional interface because it contains only one abstract method. This method takes one parameter and returns a `boolean` value. The method is so simple that it might not be worth it to define one in your application. Consequently, the JDK defines several standard functional interfaces, which you can find in the package `java.util.function`.

For example, you can use the `Predicate<T>` interface in place of `CheckPerson`. This interface contains the method `boolean test(T t)`:

```
interface Predicate<T> {
    boolean test(T t);
}
```

The interface `Predicate<T>` is an example of a generic interface. (For more information about generics, see the Generics (Updated) lesson.) Generic types (such as generic interfaces) specify one or more type parameters within angle brackets (<>). This interface contains only one type parameter, `T`. When you declare or instantiate a generic type with actual type arguments, you have a parameterized type. For example, the parameterized type `Predicate<Person>` is the following:

```
interface Predicate<Person> {
    boolean test(Person t);
}
```

This parameterized type contains a method that has the same return type and parameters as `CheckPerson.boolean test(Person p)`. Consequently, you can use `Predicate<T>` in place of

CheckPerson as the following method demonstrates:

```
public static void printPersonsWithPredicate(
    List<Person> roster, Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

As a result, the following method invocation is the same as when you invoked printPersons in Approach 3: Specify Search Criteria Code in a Local Class to obtain members who are eligible for Selective Service:

```
printPersonsWithPredicate(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

This is not the only possible place in this method to use a lambda expression. The following approach suggests other ways to use lambda expressions.

Approach 7: Use Lambda Expressions Throughout Your Application

Reconsider the method printPersonsWithPredicate to see where else you could use lambda expressions:

```
public static void printPersonsWithPredicate(
    List<Person> roster, Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

This method checks each Person instance contained in the List parameter roster whether it satisfies the criteria specified in the Predicate parameter tester. If the Person instance does satisfy the criteria specified by tester, the method printPersron is invoked on the Person instance.

Instead of invoking the method printPerson, you can specify a different action to perform on those Person instances that satisfy the criteria specified by tester. You can specify this action with a lambda expression. Suppose you want a lambda expression similar to printPerson, one that takes one argument (an object of type Person) and returns void. Remember, to use a lambda expression, you need to implement a functional interface. In this case, you need a functional interface that contains an abstract method that can take one argument of type Person and returns void. The Consumer<T> interface contains the method void accept(T t), which has these characteristics. The following method replaces the invocation p.printPerson() with an instance of Consumer<Person> that invokes the method accept:

```
public static void processPersons(
    List<Person> roster,
    Predicate<Person> tester,
    Consumer<Person> block) {
        for (Person p : roster) {
            if (tester.test(p)) {
                block.accept(p);
            }
        }
}
```

As a result, the following method invocation is the same as when you invoked `printPersons` in Approach 3: Specify Search Criteria Code in a Local Class to obtain members who are eligible for Selective Service. The lambda expression used to print members is highlighted:

```
processPersons(
     roster,
     p -> p.getGender() == Person.Sex.MALE
         && p.getAge() >= 18
         && p.getAge() <= 25,
     p -> p.printPerson()
);
```

What if you want to do more with your members' profiles than printing them out. Suppose that you want to validate the members' profiles or retrieve their contact information? In this case, you need a functional interface that contains an abstract method that returns a value. The `Function<T,R>` interface contains the method `R apply(T t)`. The following method retrieves the data specified by the parameter `mapper`, and then performs an action on it specified by the parameter `block`:

```
public static void processPersonsWithFunction(
    List<Person> roster,
    Predicate<Person> tester,
    Function<Person, String> mapper,
    Consumer<String> block) {
    for (Person p : roster) {
        if (tester.test(p)) {
            String data = mapper.apply(p);
            block.accept(data);
        }
    }
}
```

The following method retrieves the email address from each member contained in `roster` who are eligible for Selective Service and then prints it:

```
processPersonsWithFunction(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

Approach 8: Use Generics More Extensively

Reconsider the method `processPersonsWithFunction`. The following is a generic version of it that accepts, as a parameter, a collection that contains elements of any data type:

```
public static <X, Y> void processElements(
    Iterable<X> source,
    Predicate<X> tester,
    Function <X, Y> mapper,
    Consumer<Y> block) {
    for (X p : source) {
        if (tester.test(p)) {
            Y data = mapper.apply(p);
            block.accept(data);
        }
    }
}
```

To print the e-mail address of members who are eligible for Selective Service, invoke the `processElements` method as follows:

```
processElements(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

This method invocation performs the following actions:

1. Obtains a source of objects from the collection `source`. In this example, it obtains a source of `Person` objects from the collection `roster`. Notice that the collection `roster`, which is a collection of type `List`, is also an object of type `Iterable`.
2. Filters objects that match the `Predicate` object `tester`. In this example, the `Predicate` object is a lambda expression that specifies which members would be eligible for Selective Service.
3. Maps each filtered object to a value as specified by the `Function` object `mapper`. In this example, the `Function` object is a lambda expression that returns the e-mail address of a member.
4. Performs an action on each mapped object as specified by the `Consumer` object `block`. In this example, the `Consumer` object is a lambda expression that prints a string, which is the e-mail address returned by the `Function` object.

You can replace each of these actions with an aggregate operation.

Approach 9: Use Aggregate Operations That Accept Lambda Expressions as Parameters

The following example uses aggregate operations to print the e-mail addresses of those members contained in the collection `roster` who are eligible for Selective Service:

```
roster
    .stream()
    .filter(
        p -> p.getGender() == Person.Sex.MALE
            && p.getAge() >= 18
            && p.getAge() <= 25)
    .map(p -> p.getEmailAddress())
    .forEach(email -> System.out.println(email));
```

The following table maps each of the operations the method `processElements` performs with the corresponding aggregate operation:

`processElements` Action	Aggregate Operation
Obtain a source of objects	`Stream<E> stream()`
Filter objects that match a `Predicate` object	`Stream<T> filter(Predicate<? super T> predicate)`
Map objects to another value as specified by a `Function` object	`<R> Stream<R> map(Function<? super T,? extends R> mapper)`
Perform an action as specified by a `Consumer` object	`void forEach(Consumer<? super T> action)`

The operations `filter`, `map`, and `forEach` are *aggregate operations*. Aggregate operations process elements from a stream, not directly from a collection (which is the reason why the first method invoked in this example is `stream`). A *stream* is a sequence of elements. Unlike a collection, it is not a data structure that stores elements. Instead, a stream carries values from a source, such as collection, through a pipeline. A *pipeline* is a sequence of stream operations, which in this example is `filter- map-forEach`. In addition, aggregate operations typically accept lambda expressions as parameters, enabling you to customize how they behave.

For a more thorough discussion of aggregate operations, see the Aggregate Operations lesson.

Lambda Expressions in GUI Applications

To process events in a graphical user interface (GUI) application, such as keyboard actions, mouse actions, and scroll actions, you typically create event handlers, which usually involves implementing a particular interface. Often, event handler interfaces are functional interfaces; they tend to have only one method.

In the JavaFX example `HelloWorld.java` (discussed in the previous section Anonymous Classes), you can replace the highlighted anonymous class with a lambda expression in this statement:

```
        btn.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });
```

The method invocation `btn.setOnAction` specifies what happens when you select the button represented by the `btn` object. This method requires an object of type `EventHandler<ActionEvent>`. The `EventHandler<ActionEvent>` interface contains only one method, `void handle(T event)`. This interface is a functional interface, so you could use the following highlighted lambda expression to replace it:

```
        btn.setOnAction(
          event -> System.out.println("Hello World!")
        );
```

Syntax of Lambda Expressions

A lambda expression consists of the following:

- A comma-separated list of formal parameters enclosed in parentheses. The `CheckPerson.test` method contains one parameter, `p`, which represents an instance of the `Person` class.
  Note: You can omit the data type of the parameters in a lambda expression. In addition, you can omit the parentheses if there is only one parameter. For example, the following lambda expression is also valid:
  ```
  p -> p.getGender() == Person.Sex.MALE
      && p.getAge() >= 18
      && p.getAge() <= 25
  ```
- The arrow token, `->`
- A body, which consists of a single expression or a statement block. This example uses the following expression:
  ```
  p.getGender() == Person.Sex.MALE
      && p.getAge() >= 18
      && p.getAge() <= 25
  ```
  If you specify a single expression, then the Java runtime evaluates the expression and then returns its value. Alternatively, you can use a return statement:
  ```
  p -> {
      return p.getGender() == Person.Sex.MALE
          && p.getAge() >= 18
          && p.getAge() <= 25;
  }
  ```
  A return statement is not an expression; in a lambda expression, you must enclose statements in braces (`{}`). However, you do not have to enclose a void method invocation in braces. For example, the following is a valid lambda expression:
  ```
  email -> System.out.println(email)
  ```

Note that a lambda expression looks a lot like a method declaration; you can consider lambda expressions as anonymous methodsmethods without a name.

The following example, `Calculator`, is an example of lambda expressions that take more than one formal parameter:

```

public class Calculator {

    interface IntegerMath {
        int operation(int a, int b);
    }

    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }

    public static void main(String... args) {

        Calculator myApp = new Calculator();
        IntegerMath addition = (a, b) -> a + b;
        IntegerMath subtraction = (a, b) -> a - b;
        System.out.println("40 + 2 = " +
            myApp.operateBinary(40, 2, addition));
        System.out.println("20 - 10 = " +
            myApp.operateBinary(20, 10, subtraction));
    }
```

```
}
```

The method `operateBinary` performs a mathematical operation on two integer operands. The operation itself is specified by an instance of `IntegerMath`. The example defines two operations with lambda expressions, `addition` and `subtraction`. The example prints the following:

```
40 + 2 = 42
20 - 10 = 10
```

Accessing Local Variables of the Enclosing Scope

Like local and anonymous classes, lambda expressions can capture variables; they have the same access to local variables of the enclosing scope. However, unlike local and anonymous classes, lambda expressions do not have any shadowing issues (see Shadowing for more information). Lambda expressions are lexically scoped. This means that they do not inherit any names from a supertype or introduce a new level of scoping. Declarations in a lambda expression are interpreted just as they are in the enclosing environment. The following example, `LambdaScopeTest`, demonstrates this:

```

import java.util.function.Consumer;

public class LambdaScopeTest {

    public int x = 0;

    class FirstLevel {

        public int x = 1;

        void methodInFirstLevel(int x) {

            // The following statement causes the compiler to generate
            // the error "local variables referenced from a lambda expression
            // must be final or effectively final" in statement A:
            //
            // x = 99;

            Consumer<Integer> myConsumer = (y) ->
            {
                System.out.println("x = " + x); // Statement A
                System.out.println("y = " + y);
                System.out.println("this.x = " + this.x);
                System.out.println("LambdaScopeTest.this.x = " +
                    LambdaScopeTest.this.x);
            };

            myConsumer.accept(x);

        }
    }

    public static void main(String... args) {
        LambdaScopeTest st = new LambdaScopeTest();
        LambdaScopeTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
```

This example generates the following output:

```
x = 23
y = 23
this.x = 1
LambdaScopeTest.this.x = 0
```

If you substitute the parameter `x` in place of `y` in the declaration of the lambda expression `myConsumer`, then the compiler generates an error:

```
Consumer<Integer> myConsumer = (x) -> {
    // ...
}
```

The compiler generates the error "variable x is already defined in method methodInFirstLevel(int)" because the lambda expression does not introduce a new level of scoping. Consequently, you can directly access fields, methods, and local variables of the enclosing scope. For example, the lambda expression directly accesses the parameter `x` of the method `methodInFirstLevel`. To access variables in the enclosing class, use the keyword `this`. In this example, `this.x` refers to the member variable `FirstLevel.x`.

However, like local and anonymous classes, a lambda expression can only access local variables and parameters of the enclosing block that are final or effectively final. For example, suppose that you add the following assignment statement immediately after the `methodInFirstLevel` definition statement:

```
void methodInFirstLevel(int x) {
    x = 99;
    // ...
}
```

Because of this assignment statement, the variable `FirstLevel.x` is not effectively final anymore. As a result, the Java compiler generates an error message similar to "local variables referenced from a lambda expression must be final or effectively final" where the lambda expression `myConsumer` tries to access the `FirstLevel.x` variable:

```
System.out.println("x = " + x);
```

Target Typing

How do you determine the type of a lambda expression? Recall the lambda expression that selected members who are male and between the ages 18 and 25 years:

```
p -> p.getGender() == Person.Sex.MALE
    && p.getAge() >= 18
    && p.getAge() <= 25
```

This lambda expression was used in the following two methods:

- `public static void printPersons(List<Person> roster, CheckPerson tester)` in Approach 3: Specify Search Criteria Code in a Local Class
- `public void printPersonsWithPredicate(List<Person> roster, Predicate<Person> tester)` in Approach 6: Use Standard Functional Interfaces with Lambda Expressions

When the Java runtime invokes the method `printPersons`, it's expecting a data type of `CheckPerson`, so the lambda expression is of this type. However, when the Java runtime invokes the method `printPersonsWithPredicate`, it's expecting a data type of `Predicate<Person>`, so the lambda expression is of this type. The data type that these methods expect is called the *target type*. To determine the type of a lambda expression, the Java compiler uses the target type of the context or situation in which the lambda expression was found. It follows that you can only use lambda expressions in situations in which the Java compiler can determine a target type:

- Variable declarations
- Assignments
- Return statements
- Array initializers
- Method or constructor arguments
- Lambda expression bodies
- Conditional expressions, `?:`
- Cast expressions

Target Types and Method Arguments

For method arguments, the Java compiler determines the target type with two other language features: overload resolution and type argument inference.

Consider the following two functional interfaces (`java.lang.Runnable` and `java.util.concurrent.Callable<V>`):

```
public interface Runnable {
    void run();
}

public interface Callable<V> {
    V call();
}
```

The method `Runnable.run` does not return a value, whereas `Callable<V>.call` does.

Suppose that you have overloaded the method `invoke` as follows (see Defining Methods for more information about overloading methods):

```
void invoke(Runnable r) {
    r.run();
}

<T> T invoke(Callable<T> c) {
    return c.call();
}
```

Which method will be invoked in the following statement?

```
String s = invoke(() -> "done");
```

The method `invoke(Callable<T>)` will be invoked because that method returns a value; the method `invoke(Runnable)` does not. In this case, the type of the lambda expression `() -> "done"` is

`Callable<T>`.

Serialization

You can serialize a lambda expression if its target type and its captured arguments are serializable. However, like inner classes, the serialization of lambda expressions is strongly discouraged.

Method References

You use lambda expressions to create anonymous methods. Sometimes, however, a lambda expression does nothing but call an existing method. In those cases, it's often clearer to refer to the existing method by name. Method references enable you to do this; they are compact, easy-to-read lambda expressions for methods that already have a name.

Consider again the `Person` class discussed in the section Lambda Expressions:

```
public class Person {

    public enum Sex {
        MALE, FEMALE
    }

    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    public int getAge() {
        // ...
    }

    public Calendar getBirthday() {
        return birthday;
    }

    public static int compareByAge(Person a, Person b) {
        return a.birthday.compareTo(b.birthday);
    }}
```

Suppose that the members of your social networking application are contained in an array, and you want to sort the array by age. You could use the following code (find the code excerpts described in this section in the example `MethodReferencesTest`):

```
Person[] rosterAsArray = roster.toArray(new Person[roster.size()]);

class PersonAgeComparator implements Comparator<Person> {
    public int compare(Person a, Person b) {
        return a.getBirthday().compareTo(b.getBirthday());
    }
}

Arrays.sort(rosterAsArray, new PersonAgeComparator());
```

The method signature of this invocation of `sort` is the following:

```
static <T> void sort(T[] a, Comparator<? super T> c)
```

Notice that the interface `Comparator` is a functional interface. Therefore, you could use a lambda expression instead of defining and then creating a new instance of a class that implements `Comparator`:

```
Arrays.sort(rosterAsArray,
    (Person a, Person b) -> {
        return a.getBirthday().compareTo(b.getBirthday());
    }
);
```

However, this method to compare the birth dates of two `Person` instances already exists as `Person.compareByAge`. You can invoke this method instead in the body of the lambda expression:

```
Arrays.sort(rosterAsArray,
    (a, b) -> Person.compareByAge(a, b)
);
```

Because this lambda expression invokes an existing method, you can use a method reference instead of a lambda expression:

```
Arrays.sort(rosterAsArray, Person::compareByAge);
```

The method reference `Person::compareByAge` is semantically the same as the lambda expression `(a, b) -> Person.compareByAge(a, b)`. Each has the following characteristics:

- Its formal parameter list is copied from `Comparator<Person>.compare`, which is `(Person, Person)`.
- Its body calls the method `Person.compareByAge`.

Kinds of Method References

There are four kinds of method references:

Kind	Example
Reference to a static method	`ContainingClass::staticMethodName`
Reference to an instance method of a particular object	`ContainingObject::instanceMethodName`
Reference to an instance method of an arbitrary object of a particular type	`ContainingType::methodName`
Reference to a constructor	`ClassName::new`

Reference to a Static Method

The method reference `Person::compareByAge` is a reference to a static method.

Reference to an Instance Method of a Particular Object

The following is an example of a reference to an instance method of a particular object:

```
class ComparisonProvider {
    public int compareByName(Person a, Person b) {
        return a.getName().compareTo(b.getName());
    }

    public int compareByAge(Person a, Person b) {
        return a.getBirthday().compareTo(b.getBirthday());
    }
}
```

```
ComparisonProvider myComparisonProvider = new ComparisonProvider();
Arrays.sort(rosterAsArray, myComparisonProvider::compareByName);
```

The method reference `myComparisonProvider::compareByName` invokes the method `compareByName` that is part of the object `myComparisonProvider`. The JRE infers the method type arguments, which in this case are `(Person, Person)`.

Reference to an Instance Method of an Arbitrary Object of a Particular Type

The following is an example of a reference to an instance method of an arbitrary object of a particular type:

```
String[] stringArray = { "Barbara", "James", "Mary", "John",
    "Patricia", "Robert", "Michael", "Linda" };
Arrays.sort(stringArray, String::compareToIgnoreCase);
```

The equivalent lambda expression for the method reference `String::compareToIgnoreCase` would have the formal parameter list `(String a, String b)`, where `a` and `b` are arbitrary names used to better describe this example. The method reference would invoke the method `a.compareToIgnoreCase(b)`.

Reference to a Constructor

You can reference a constructor in the same way as a static method by using the name `new`. The following method copies elements from one collection to another:

```
public static <T, SOURCE extends Collection<T>, DEST extends Collection<T>>
    DEST transferElements(
        SOURCE sourceCollection,
        Supplier<DEST> collectionFactory) {

        DEST result = collectionFactory.get();
        for (T t : sourceCollection) {
            result.add(t);
        }
        return result;
}
```

The functional interface `Supplier` contains one method `get` that takes no arguments and returns an object. Consequently, you can invoke the method `transferElements` with a lambda expression as follows:

```
Set<Person> rosterSetLambda =
    transferElements(roster, () -> { return new HashSet<>(); });
```

You can use a constructor reference in place of the lambda expression as follows:

```
Set<Person> rosterSet = transferElements(roster, HashSet::new);
```

The Java compiler infers that you want to create a `HashSet` collection that contains elements of type `Person`. Alternatively, you can specify this as follows:

```
Set<Person> rosterSet = transferElements(roster, HashSet<Person>::new);
```

When to Use Nested Classes, Local Classes, Anonymous Classes, and Lambda Expressions

As mentioned in the section Nested Classes, nested classes enable you to logically group classes that are only used in one place, increase the use of encapsulation, and create more readable and maintainable code. Local classes, anonymous classes, and lambda expressions also impart these advantages; however, they are intended to be used for more specific situations:

- Local class: Use it if you need to create more than one instance of a class, access its constructor, or introduce a new, named type (because, for example, you need to invoke additional methods later).
- Anonymous class: Use it if you need to declare fields or additional methods.
- Lambda expression:
    - Use it if you are encapsulating a single unit of behavior that you want to pass to other code. For example, you would use a lambda expression if you want a certain action performed on each element of a collection, when a process is completed, or when a process encounters an error.
    - Use it if you need a simple instance of a functional interface and none of the preceding criteria apply (for example, you do not need a constructor, a named type, fields, or additional methods).
- Nested class: Use it if your requirements are similar to those of a local class, you want to make the type more widely available, and you don't require access to local variables or method parameters.
    - Use a non-static nested class (or inner class) if you require access to an enclosing instance's non-public fields and methods. Use a static nested class if you don't require this access.

Questions and Exercises: Nested Classes

Questions

1. The program `Problem.java` doesn't compile. What do you need to do to make it compile? Why?
2. Use the Java API documentation for the `Box` class (in the `javax.swing` package) to help you answer the following questions.
a. What static nested class does `Box` define?
b. What inner class does `Box` define?
c. What is the superclass of `Box`'s inner class?
d. Which of `Box`'s nested classes can you use from any class?
e. How do you create an instance of `Box`'s `Filler` class?

Exercises

1. Get the file `Class1.java`. Compile and run `Class1`. What is the output?
2. The following exercises involve modifying the class `DataStructure.java`, which the section Inner Class Example discusses.
a. Define a method named `print(DataStructureIterator iterator)`. Invoke this method with an instance of the class `EvenIterator` so that it performs the same function as the method `printEven`.
b. Invoke the method `print(DataStructureIterator iterator)` so that it prints elements that have an odd index value. Use an anonymous class as the method's argument instead of an instance of the interface `DataStructureIterator`.
c. Define a method named `print(java.util.Function<Integer, Boolean> iterator)` that performs the same function as `print(DataStructureIterator iterator)`. Invoke this method with a lambda expression to print elements that have an even index value. Invoke this method again with a lambda expression to print elements that have an odd index value.
d. Define two methods so that the following two statements print elements that have an even index value and elements that have an odd index value:

```
DataStructure ds = new DataStructure()
// ...
ds.print(DataStructure::isEvenIndex);
ds.print(DataStructure::isOddIndex);
```

Check your answers.

Questions

1. Question: The program `Problem.java` doesn't compile. What do you need to do to make it compile? Why?
Answer: Delete `static` in front of the declaration of the `Inner` class. An static inner class does not have access to the instance fields of the outer class. See `ProblemSolved.java`.
2. Use the Java API documentation for the [`Box`] class (in the `javax.swing` package) to help you answer the following questions.
a. Question: What static nested class does `Box` define?
Answer: `Box.Filler`
b. Question: What inner class does `Box` define?
Answer: `Box.AccessibleBox`
c. Question: What is the superclass of `Box`'s inner class?
Answer: `[java.awt.]Container.AccessibleAWTContainer`
d. Question: Which of `Box`'s nested classes can you use from any class?
Answer: `Box.Filler`
e. Question: How do you create an instance of `Box`'s `Filler` class?
Answer: `new Box.Filler(minDimension, prefDimension, maxDimension)`

Exercises

1. Exercise: Get the file `Class1.java`. Compile and run `Class1`. What is the output?
Answer:
```
InnerClass1: getString invoked.
InnerClass1: getAnotherString invoked.
```
2. Exercise: The following exercises involve modifying the class `DataStructure.java`, which the section [Inner Class Example] discusses.
a. Define a method named `print(DataStructureIterator iterator)`. Invoke this method with an instance of the class `EvenIterator` so that it performs the same function as the method `printEven`.
Hint: These statements do not compile if you specify them in the `main` method:
```
DataStructure ds = new DataStructure();
ds.print(new EvenIterator());
```
The compiler generates the error message, "non-static variable this cannot be referenced from a static context" when it encounters the expression `new EvenIterator()`. The class `EvenIterator` is an inner, non-static class. This means that you can create an instance of `EvenIterator` only inside an instance of the outer class, `DataStructure`.
You can define a method in `DataStructure` that creates and returns a new instance of `EvenIterator`.
b. Invoke the method `print(DataStructureIterator iterator)` so that it prints elements that have an odd index value. Use an anonymous class as the method's argument instead of an instance of the interface `DataStructureIterator`.
Hint: You cannot access the private members `SIZE` and `arrayOfInts` outside the class `DataStructure`, which means that you cannot access these private members from an anonymous class defined outside `DataStructure`.
You can define methods that access the private members `SIZE` and `arrayOfInts` and then use them in your anonymous class.

c. Define a method named `print(java.util.Function<Integer, Boolean> iterator)` that performs the same function as `print(DataStructureIterator iterator)`. Invoke this method with a lambda expression to print elements that have an even index value. Invoke this method again with a lambda expression to print elements that have an odd index value.
Hint: In this `print` method, you can step though the elements contained in the array `arrayOfInts` with a `for` expression. For each index value, invoke the method `function.apply`. If this method returns a true value for a particular index value, print the element contained in that index value.
To invoke this `print` method to print elements that have an even index value, you can specify a lambda expression that implements the method `Boolean Function.apply(Integer t)`. This lambda expression takes one `Integer` argument (the index) and returns a `Boolean` value (`Boolean.TRUE` if the index value is even, `Boolean.FALSE` otherwise).
d. Define two methods so that these statements print elements that have an even index value and then elements that have an odd index value:

```
DataStructure ds = new DataStructure()
// ...
ds.print(DataStructure::isEvenIndex);
ds.print(DataStructure::isOddIndex);
```

Hint: Create two methods in the class `DataStructure` named `isEvenIndex` and `isOddIndex` that have the same parameter list and return type as the abstract method `Boolean Function<Integer, Boolean>.apply(Integer t)`. This means that the methods take one `Integer` argument (the index) and return a `Boolean` value.
Answer: See the file `DataStructure.java`.

Enum Types

An *enum type* is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it. Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week.

Because they are constants, the names of an enum type's fields are in uppercase letters.

In the Java programming language, you define an enum type by using the `enum` keyword. For example, you would specify a days-of-the-week enum type as:

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

You should use enum types any time you need to represent a fixed set of constants. That includes natural enum types such as the planets in our solar system and data sets where you know all possible values at compile time—for example, the choices on a menu, command line flags, and so on.

Here is some code that shows you how to use the `Day` enum defined above:

```
public class EnumTest {
    Day day;

    public EnumTest(Day day) {
        this.day = day;
    }

    public void tellItLikeItIs() {
        switch (day) {
            case MONDAY:
                System.out.println("Mondays are bad.");
                break;

            case FRIDAY:
                System.out.println("Fridays are better.");
                break;

            case SATURDAY: case SUNDAY:
                System.out.println("Weekends are best.");
                break;

            default:
                System.out.println("Midweek days are so-so.");
                break;
        }
    }

    public static void main(String[] args) {
        EnumTest firstDay = new EnumTest(Day.MONDAY);
        firstDay.tellItLikeItIs();
        EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);
        thirdDay.tellItLikeItIs();
        EnumTest fifthDay = new EnumTest(Day.FRIDAY);
        fifthDay.tellItLikeItIs();
        EnumTest sixthDay = new EnumTest(Day.SATURDAY);
        sixthDay.tellItLikeItIs();
        EnumTest seventhDay = new EnumTest(Day.SUNDAY);
        seventhDay.tellItLikeItIs();
    }
}
```

The output is:

```
Mondays are bad.
Midweek days are so-so.
Fridays are better.
Weekends are best.
Weekends are best.
```

Java programming language enum types are much more powerful than their counterparts in other languages. The `enum` declaration defines a *class* (called an *enum type*). The enum class body can include methods and other fields. The compiler automatically adds some special methods when it creates an enum. For example, they have a static `values` method that returns an array containing all of the values of the enum in the order they are declared. This method is commonly used in combination with the for-each construct to iterate over the values of an enum type. For example, this code from the `Planet` class example below iterates over all the planets in the solar system.

```
for (Planet p : Planet.values()) {
    System.out.printf("Your weight on %s is %f%n",
                      p, p.surfaceWeight(mass));
}
```

Note:*All* enums implicitly extend `java.lang.Enum`. Since Java does not support multiple inheritance, an enum cannot extend anything else.

In the following example, `Planet` is an enum type that represents the planets in the solar system. They are defined with constant mass and radius properties.

Each enum constant is declared with values for the mass and radius parameters. These values are passed to the constructor when the constant is created. Java requires that the constants be defined first, prior to any fields or methods. Also, when there are fields and methods, the list of enum constants must end with a semicolon.

Note: The constructor for an enum type must be package-private or private access. It automatically creates the constants that are defined at the beginning of the enum body. You cannot invoke an enum constructor yourself.

In addition to its properties and constructor, `Planet` has methods that allow you to retrieve the surface gravity and weight of an object on each planet. Here is a sample program that takes your weight on earth (in any unit) and calculates and prints your weight on all of the planets (in the same unit):

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);

    private final double mass;   // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    private double mass() { return mass; }
    private double radius() { return radius; }

    // universal gravitational constant  (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage: java Planet <earth_weight>");
            System.exit(-1);
        }
        double earthWeight = Double.parseDouble(args[0]);
        double mass = earthWeight/EARTH.surfaceGravity();
        for (Planet p : Planet.values())
           System.out.printf("Your weight on %s is %f%n",
                             p, p.surfaceWeight(mass));
    }
}
```

If you run `Planet.class` from the command line with an argument of 175, you get this output:

```
$ java Planet 175
Your weight on MERCURY is 66.107583
Your weight on VENUS is 158.374842
Your weight on EARTH is 175.000000
Your weight on MARS is 66.279007
Your weight on JUPITER is 442.847567
Your weight on SATURN is 186.552719
Your weight on URANUS is 158.397260
Your weight on NEPTUNE is 199.207413
```

Questions and Exercises: Enum Types

Questions

1. True or false: an `Enum` type can be a subclass of `java.lang.String`.

Exercises

1. Rewrite the class `Card` from the exercise in Questions and Exercises: Classes so that it represents the rank and suit of a card with enum types.
2. Rewrite the `Deck` class.

Check your answers.

Questions

1. Question: True or false: an `Enum` type can be a subclass of `java.lang.String`. **Answer**: All enums implicitly extend `java.lang.Enum`. Since Java does not support multiple inheritance, an enum cannot extend anything else.

Exercises

1. Exercise: Rewrite the class `Card` from the exercise in Questions and Exercises: Classes so that it represents the rank and suit of a card with enum types.
Answer: See `Card3.java`, `Suit.java`, and `Rank.java`.
2. Exercise: Rewrite the `Deck` class.
Answer: See `Deck3.java`.

Lesson: Annotations

Annotations, a form of metadata, provide data about a program that is not part of the program itself. Annotations have no direct effect on the operation of the code they annotate.

Annotations have a number of uses, among them:

- **Information for the compiler** — Annotations can be used by the compiler to detect errors or suppress warnings.
- **Compile-time and deployment-time processing** — Software tools can process annotation information to generate code, XML files, and so forth.
- **Runtime processing** — Some annotations are available to be examined at runtime.

This lesson explains where annotations can be used, how to apply annotations, what predefined annotation types are available in the Java Platform, Standard Edition (Java SE API), how type annnotations can be used in conjunction with pluggable type systems to write code with stronger type checking, and how to implement repeating annotations.

Note: See online version of topics in this ebook to download complete source code.

Annotations Basics

The Format of an Annotation

In its simplest form, an annotation looks like the following:

```
@Entity
```

The at sign character (`@`) indicates to the compiler that what follows is an annotation. In the following example, the annotation's name is `Override`:

```
@Override
void mySuperMethod() { ... }
```

The annotation can include *elements*, which can be named or unnamed, and there are values for those elements:

```
@Author(
   name = "Benjamin Franklin",
   date = "3/27/2003"
)
class MyClass() { ... }
```

or

```
@SuppressWarnings(value = "unchecked")
void myMethod() { ... }
```

If there is just one element named `value`, then the name can be omitted, as in:

```
@SuppressWarnings("unchecked")
void myMethod() { ... }
```

If the annotation has no elements, then the parentheses can be omitted, as shown in the previous `@Override` example.

It is also possible to use multiple annotations on the same declaration:

```
@Author(name = "Jane Doe")
@EBook
class MyClass { ... }
```

If the annotations have the same type, then this is called a repeating annotation:

```
@Author(name = "Jane Doe")
```

```
@Author(name = "John Smith")
class MyClass { ... }
```

Repeating annotations are supported as of the Java SE 8 release. For more information, see Repeating Annotations.

The annotation type can be one of the types that are defined in the `java.lang` or `java.lang.annotation` packages of the Java SE API. In the previous examples, `Override` and `SuppressWarnings` are predefined Java annotations. It is also possible to define your own annotation type. The `Author` and `Ebook` annotations in the previous example are custom annotation types.

Where Annotations Can Be Used

Annotations can be applied to declarations: declarations of classes, fields, methods, and other program elements. When used on a declaration, each annotation often appears, by convention, on its own line.

As of the Java SE 8 release, annotations can also be applied to the *use* of types. Here are some examples:

- Class instance creation expression:

  ```
  new @Interned MyObject();
  ```

- Type cast:

  ```
  myString = (@NonNull String) str;
  ```

- `implements` clause:

  ```
  class UnmodifiableList<T> implements
      @Readonly List<@Readonly T> { ... }
  ```

- Thrown exception declaration:

  ```
  void monitorTemperature() throws
      @Critical TemperatureException { ... }
  ```

This form of annotation is called a *type annotation*. For more information, see Type Annotations and Pluggable Type Systems.

Declaring an Annotation Type

Many annotations replace comments in code.

Suppose that a software group traditionally starts the body of every class with comments providing important information:

```
public class Generation3List extends Generation2List {

   // Author: John Doe
   // Date: 3/17/2002
   // Current revision: 6
   // Last modified: 4/12/2004
   // By: Jane Doe
   // Reviewers: Alice, Bill, Cindy

   // class code goes here

}
```

To add this same metadata with an annotation, you must first define the *annotation type*. The syntax for doing this is:

```
@interface ClassPreamble {
   String author();
   String date();
   int currentRevision() default 1;
   String lastModified() default "N/A";
   String lastModifiedBy() default "N/A";
   // Note use of array
   String[] reviewers();
}
```

The annotation type definition looks similar to an interface definition where the keyword `interface` is preceded by the at sign (`@`) (@ = AT, as in annotation type). Annotation types are a form of *interface*, which will be covered in a later lesson. For the moment, you do not need to understand interfaces.

The body of the previous annotation definition contains *annotation type element* declarations, which look a lot like methods. Note that they can define optional default values.

After the annotation type is defined, you can use annotations of that type, with the values filled in, like this:

```
@ClassPreamble (
   author = "John Doe",
   date = "3/17/2002",
   currentRevision = 6,
   lastModified = "4/12/2004",
   lastModifiedBy = "Jane Doe",
   // Note array notation
   reviewers = {"Alice", "Bob", "Cindy"}
)
public class Generation3List extends Generation2List {
```

```
// class code goes here

}
```

Note: To make the information in `@ClassPreamble` appear in Javadoc-generated documentation, you must annotate the `@ClassPreamble` definition with the `@Documented` annotation:

```
// import this to use @Documented
import java.lang.annotation.*;

@Documented
@interface ClassPreamble {

   // Annotation element definitions

}
```

Predefined Annotation Types

A set of annotation types are predefined in the Java SE API. Some annotation types are used by the Java compiler, and some apply to other annotations.

Annotation Types Used by the Java Language

The predefined annotation types defined in `java.lang` are `@Deprecated`, `@Override`, and `@SuppressWarnings`.

@Deprecated `@Deprecated` annotation indicates that the marked element is *deprecated* and should no longer be used. The compiler generates a warning whenever a program uses a method, class, or field with the `@Deprecated` annotation. When an element is deprecated, it should also be documented using the Javadoc `@deprecated` tag, as shown in the following example. The use of the at sign (`@`) in both Javadoc comments and in annotations is not coincidental: they are related conceptually. Also, note that the Javadoc tag starts with a lowercase *d* and the annotation starts with an uppercase *D*.

```
   // Javadoc comment follows
    /**
     * @deprecated
     * explanation of why it was deprecated
     */
    @Deprecated
    static void deprecatedMethod() { }
}
```

@Override `@Override` annotation informs the compiler that the element is meant to override an element declared in a superclass. Overriding methods will be discussed in Interfaces and Inheritance.

```
   // mark method as a superclass method
   // that has been overridden
   @Override
   int overriddenMethod() { }
```

While it is not required to use this annotation when overriding a method, it helps to prevent errors. If a method marked with `@Override` fails to correctly override a method in one of its superclasses, the compiler generates an error.

@SuppressWarnings `@SuppressWarnings` annotation tells the compiler to suppress specific warnings that it would otherwise generate. In the following example, a deprecated method is used, and the compiler usually generates a warning. In this case, however, the annotation causes the warning to be suppressed.

```
   // use a deprecated method and tell
   // compiler not to generate a warning
   @SuppressWarnings("deprecation")
    void useDeprecatedMethod() {
        // deprecation warning
        // - suppressed
        objectOne.deprecatedMethod();
    }
```

Every compiler warning belongs to a category. The Java Language Specification lists two categories: `deprecation` and `unchecked`. The `unchecked` warning can occur when interfacing with legacy code written before the advent of generics. To suppress multiple categories of warnings, use the following syntax:

```
@SuppressWarnings({"unchecked", "deprecation"})
```

@SafeVarargs `@SafeVarargs` annotation, when applied to a method or constructor, asserts that the code does not perform potentially unsafe operations on its `varargs` parameter. When this annotation type is used, unchecked warnings relating to `varargs` usage are suppressed.

@FunctionalInterface `@FunctionalInterface` annotation, introduced in Java SE 8, indicates that the type declaration is intended to be a functional interface, as defined by the Java Language Specification.

Annotations That Apply to Other Annotations

Annotations that apply to other annotations are called *meta-annotations*. There are several meta-annotation types defined in `java.lang.annotation`.

@Retention `@Retention` annotation specifies how the marked annotation is stored:

- `RetentionPolicy.SOURCE` The marked annotation is retained only in the source level and is ignored by the compiler.
- `RetentionPolicy.CLASS` The marked annotation is retained by the compiler at compile time, but is ignored by the Java Virtual Machine (JVM).
- `RetentionPolicy.RUNTIME` The marked annotation is retained by the JVM so it can be used by the runtime environment.

@Documented `@Documented` annotation indicates that whenever the specified annotation is used those elements should be documented using the Javadoc tool. (By default, annotations are not included in Javadoc.) For more information, see the Javadoc tools page.

@Target `@Target` annotation marks another annotation to restrict what kind of Java elements the annotation can be applied to. A target annotation specifies one of the following element types as its value:

- `ElementType.ANNOTATION_TYPE` can be applied to an annotation type.
- `ElementType.CONSTRUCTOR` can be applied to a constructor.
- `ElementType.FIELD` can be applied to a field or property.
- `ElementType.LOCAL_VARIABLE` can be applied to a local variable.
- `ElementType.METHOD` can be applied to a method-level annotation.

- `ElementType.PACKAGE` can be applied to a package declaration.
- `ElementType.PARAMETER` can be applied to the parameters of a method.
- `ElementType.TYPE` can be applied to any element of a class.

@Inherited `@Inherited` annotation indicates that the annotation type can be inherited from the super class. (This is not true by default.) When the user queries the annotation type and the class has no annotation for this type, the class' superclass is queried for the annotation type. This annotation applies only to class declarations.

@Repeatable `@Repeatable` annotation, introduced in Java SE 8, indicates that the marked annotation can be applied more than once to the same declaration or type use. For more information, see Repeating Annotations.

Type Annotations and Pluggable Type Systems

Before the Java SE 8 release, annotations could only be applied to declarations. As of the Java SE 8 release, annotations can also be applied to any *type use*. This means that annotations can be used anywhere you use a type. A few examples of where types are used are class instance creation expressions (`new`), casts, `implements` clauses, and `throws` clauses. This form of annotation is called a *type annotation* and several examples are provided in Annotations Basics.

Type annotations were created to support improved analysis of Java programs way of ensuring stronger type checking. The Java SE 8 release does not provide a type checking framework, but it allows you to write (or download) a type checking framework that is implemented as one or more pluggable modules that are used in conjunction with the Java compiler.

For example, you want to ensure that a particular variable in your program is never assigned to null; you want to avoid triggering a `NullPointerException`. You can write a custom plug-in to check for this. You would then modify your code to annotate that particular variable, indicating that it is never assigned to null. The variable declaration might look like this:

```
@NonNull String str;
```

When you compile the code, including the `NonNull` module at the command line, the compiler prints a warning if it detects a potential problem, allowing you to modify the code to avoid the error. After you correct the code to remove all warnings, this particular error will not occur when the program runs.

You can use multiple type-checking modules where each module checks for a different kind of error. In this way, you can build on top of the Java type system, adding specific checks when and where you want them.

With the judicious use of type annotations and the presence of pluggable type checkers, you can write code that is stronger and less prone to error.

In many cases, you do not have to write your own type checking modules. There are third parties who have done the work for you. For example, you might want to take advantage of the Checker Framework created by the University of Washington. This framework includes a `NonNull` module, as well as a regular expression module, and a mutex lock module. For more information, see the Checker Framework.

Repeating Annotations

There are some situations where you want to apply the same annotation to a declaration or type use. As of the Java SE 8 release, *repeating annotations* enable you to do this.

For example, you are writing code to use a timer service that enables you to run a method at a given time or on a certain schedule, similar to the UNIX `cron` service. Now you want to set a timer to run a method, `doPeriodicCleanup`, on the last day of the month and on every Friday at 11:00 p.m. To set the timer to run, create an `@Schedule` annotation and apply it twice to the `doPeriodicCleanup` method. The first use specifies the last day of the month and the second specifies Friday at 11p.m., as shown in the following code example:

```
@Schedule(dayOfMonth="last")
@Schedule(dayOfWeek="Fri", hour="23")
public void doPeriodicCleanup() { ... }
```

The previous example applies an annotation to a method. You can repeat an annotation anywhere that you would use a standard annotation. For example, you have a class for handling unauthorized access exceptions. You annotate the class with one `@Alert` annotation for managers and another for admins:

```
@Alert(role="Manager")
@Alert(role="Administrator")
public class UnauthorizedAccessException extends SecurityException { ... }
```

For compatibility reasons, repeating annotations are stored in a *container annotation* that is automatically generated by the Java compiler. In order for the compiler to do this, two declarations are required in your code.

Step 1: Declare a Repeatable Annotation Type

The annotation type must be marked with the `@Repeatable` meta-annotation. Given the custom `Schedule` example:

```
public @interface Schedule { ... }
```

The code looks like the following:

```
@Repeatable(Schedules.class)
public @interface Schedule { ... }
```

The value of the `@Repeatable` meta-annotation, in parentheses, is the type of the container annotation that the Java compiler generates to store repeating annotations. In this example, the containing annotation type is `Schedules`, so repeating `@Schedule` annotations is stored in an `@Schedules` annotation.

Applying the same annotation to a declaration without first declaring it to be repeatable results in a compile-time error.

Step 2: Declare the Containing Annotation Type

The containing annotation type must have a `value` element with an array type. The component type of the array type must be the repeatable annotation type. The declaration for the `Schedules` containing annotation type is the following:

```
public @interface Schedules {
    Schedule[] value;
}
```

Retrieving Annotations

There are several methods available in the Reflection API that can be used to retrieve annotations. The behavior of the methods that return a single annotation, such as AnnotatedElement.getAnnotationByType(Class<T>), are unchanged in that they only return a single annotation if *one* annotation of the requested type is present. If more than one annotation of the requested type is present, you can obtain them by first getting their container annotation. In this way, legacy code continues to work. Other methods were introduced in Java SE 8 that scan through the container annotation to return multiple annotations at once, such as AnnotatedElement.getAnnotations(Class<T>). See the AnnotatedElement class specification for information on all of the available methods.

Design Considerations

When designing an annotation type, you must consider the *cardinality* of annotations of that type. It is now possible to use an annotation zero times, once, or, if the annotation's type is marked as `@Repeatable`, more than once. It is also possible to restrict where an annotation type can be used by using the `@Target` meta-annotation. For example, you can create a repeatable annotation type that can only be used on methods and fields. It is important to design your annotation type carefully to ensure that the programmer *using* the annotation finds it to be as flexible and powerful as possible.

Questions and Exercises: Annotations

Questions

1. What is wrong with the following interface?

```
public interface House {
    @Deprecated
    void open();
    void openFrontDoor();
    void openBackDoor();
}
```

2. Consider this implementation of the `House` interface, shown in Question 1.

```
public class MyHouse implements House {
    public void open() {}
    public void openFrontDoor() {}
    public void openBackDoor() {}
}
```

If you compile this program, the compiler produces an error because `open` was deprecated (in the interface). What can you do to get rid of that warning?

3. Will the following code compile without error? Why or why not?

```
public @interface Meal { ... }

@Meal("breakfast", mainDish="cereal")
@Meal("lunch", mainDish="pizza")
@Meal("dinner", mainDish="salad")
public void evaluateDiet() { ... }
```

Exercises

1. Define an annotation type for an enhancement request with elements `id`, `synopsis`, `engineer`, and `date`. Specify the default value as `unassigned` for engineer and `unknown` for date.

Check your answers.

Questions

1. Question: What is wrong with the following interface:

```
public interface House {
    @Deprecated
    public void open();
    public void openFrontDoor();
    public void openBackDoor();
}
```

Answer The documentation should reflect why `open` is deprecated and what to use instead. For example:

```
public interface House {
    /**
     * @deprecated use of open
     * is discouraged, use
     * openFrontDoor or
     * openBackDoor instead.
     */
    @Deprecated
    public void open();
    public void openFrontDoor();
    public void openBackDoor();
}
```

2. Question: Consider this implementation of the `House` interface, shown in Question 1.

```
public class MyHouse implements House {
    public void open() {}
    public void openFrontDoor() {}
    public void openBackDoor() {}
}
```

If you compile this program, the compiler produces an error because `open` was deprecated (in the interface). What can you do to get rid of that warning?

Answer: You can deprecate the implementation of `open`:

```
public class MyHouse implements House {
    // The documentation is
    // inherited from the interface.
    @Deprecated
    public void open() {}
    public void openFrontDoor() {}
    public void openBackDoor() {}
}
```

Alternatively, you can suppress the warning:

```
public class MyHouse implements House {
    @SuppressWarnings("deprecation")
    public void open() {}
    public void openFrontDoor() {}
    public void openBackDoor() {}
}
```

3. Will the following code compile without error? Why or why not?

```
public @interface Meal { ... }

@Meal("breakfast", mainDish="cereal")
@Meal("lunch", mainDish="pizza")
@Meal("dinner", mainDish="salad")
public void evaluateDiet() { ... }
```

Answer: The code fails to compile. Before JDK 8, repeatable annotations are not supported. As of JDK 8, the code fails to compile because the `Meal` annotation type was not defined to be repeatable. It can be fixed by adding the `@Repeatable` meta-annotation and defining a container annotation type:

```
@Repeatable(MealContainer.class)
public @interface Meal { ... }

public @interface MealContainer {
    Meal[] value;
}
```

Exercises

1. Exercise: Define an annotation type for an enhancement request with elements `id`, `synopsis`, `engineer`, and `date`. Specify the default value as `unassigned` for engineer and `unknown` for date.
Answer:

```
/**
 * Describes the Request-for-Enhancement (RFE) annotation type.
 */
public @interface RequestForEnhancement {
    int id();
    String synopsis();
    String engineer() default "[unassigned]";
    String date() default "[unknown]";
}
```

Lesson: Interfaces and Inheritance

Interfaces

You saw an example of implementing an interface in the previous lesson. You can read more about interfaces here—what they are for, why you might want to write one, and how to write one.

Inheritance

This section describes the way in which you can derive one class from another. That is, how a *subclass* can inherit fields and methods from a *superclass*. You will learn that all classes are derived from the `Object` class, and how to modify the methods that a subclass inherits from superclasses. This section also covers interface-like *abstract classes*.

Note: See online version of topics in this ebook to download complete source code.

Interfaces

There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, *interfaces* are such contracts.

For example, imagine a futuristic society where computer-controlled robotic cars transport passengers through city streets without a human operator. Automobile manufacturers write software (Java, of course) that operates the automobile—stop, start, accelerate, turn left, and so forth. Another industrial group, electronic guidance instrument manufacturers, make computer systems that receive GPS (Global Positioning System) position data and wireless transmission of traffic conditions and use that information to drive the car.

The auto manufacturers must publish an industry-standard interface that spells out in detail what methods can be invoked to make the car move (any car, from any manufacturer). The guidance manufacturers can then write software that invokes the methods described in the interface to command the car. Neither industrial group needs to know *how* the other group's software is implemented. In fact, each group considers its software highly proprietary and reserves the right to modify it at any time, as long as it continues to adhere to the published interface.

Interfaces in Java

In the Java programming language, an *interface* is a reference type, similar to a class, that can contain *only* constants, method signatures, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods. Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces. Extension is discussed later in this lesson.

Defining an interface is similar to creating a new class:

```
public interface OperateCar {

   // constant declarations, if any

   // method signatures

   // An enum with values RIGHT, LEFT
   int turn(Direction direction,
            double radius,
            double startSpeed,
            double endSpeed);
   int changeLanes(Direction direction,
                   double startSpeed,
                   double endSpeed);
   int signalTurn(Direction direction,
                  boolean signalOn);
   int getRadarFront(double distanceToCar,
                     double speedOfCar);
   int getRadarRear(double distanceToCar,
                    double speedOfCar);
         ......
   // more method signatures
}
```

Note that the method signatures have no braces and are terminated with a semicolon.

To use an interface, you write a class that *implements* the interface. When an instantiable class implements an interface, it provides a method body for each of the methods declared in the interface. For example,

```
public class OperateBMW760i implements OperateCar {

    // the OperateCar method signatures, with implementation --
    // for example:
    int signalTurn(Direction direction, boolean signalOn) {
       // code to turn BMW's LEFT turn indicator lights on
       // code to turn BMW's LEFT turn indicator lights off
       // code to turn BMW's RIGHT turn indicator lights on
       // code to turn BMW's RIGHT turn indicator lights off
    }

    // other members, as needed -- for example, helper classes not
    // visible to clients of the interface
}
```

In the robotic car example above, it is the automobile manufacturers who will implement the interface. Chevrolet's implementation will be substantially different from that of Toyota, of course, but both manufacturers will adhere to the same interface. The guidance manufacturers, who are the clients of the interface, will build systems that use GPS data on a car's location, digital street maps, and traffic data to drive the car. In so doing, the guidance systems will invoke the interface methods: turn, change lanes, brake, accelerate, and so forth.

Interfaces as APIs

The robotic car example shows an interface being used as an industry standard *Application Programming Interface (API)*. APIs are also common in commercial software products. Typically, a company sells a software package that contains complex methods that another company wants to use in its own software product. An example would be a package of digital image processing methods that are sold to companies making end-user graphics programs. The image processing company writes its classes to implement an interface, which it makes public to its customers. The graphics company then invokes the image processing methods using the signatures and return types defined in the interface. While the image processing company's API is made public (to its customers), its implementation of the API is kept as a closely guarded secret—in fact, it may revise the implementation at a later date as long as it continues to implement the original interface that its customers have relied on.

Defining an Interface

An interface declaration consists of modifiers, the keyword `interface`, the interface name, a comma-separated list of parent interfaces (if any), and the interface body. For example:

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {

    // constant declarations

    // base of natural logarithms
    double E = 2.718282;

    // method signatures
    void doSomething (int i, double x);
    int doSomethingElse(String s);
}
```

The `public` access specifier indicates that the interface can be used by any class in any package. If you do not specify that the interface is public, then your interface is accessible only to classes defined in the same package as the interface.

An interface can extend other interfaces, just as a class subclass or extend another class. However, whereas a class can extend only one other class, an interface can extend any number of interfaces. The interface declaration includes a comma-separated list of all the interfaces that it extends.

The Interface Body

The interface body can contain abstract methods, default methods, and static methods. An abstract method within an interface is followed by a semicolon, but no braces (an abstract method does not contain an implementation). Default methods are defined with the `default` modifier, and static methods with the `static` keyword. All abstract, default, and static methods in an interface are implicitly `public`, so you can omit the `public` modifier.

In addition, an interface can contain constant declarations. All constant values defined in an interface are implicitly `public`, `static`, and `final`. Once again, you can omit these modifiers.

Implementing an Interface

To declare a class that implements an interface, you include an `implements` clause in the class declaration. Your class can implement more than one interface, so the `implements` keyword is followed by a comma-separated list of the interfaces implemented by the class. By convention, the `implements` clause follows the `extends` clause, if there is one.

A Sample Interface, Relatable

Consider an interface that defines how to compare the size of objects.

```
public interface Relatable {

    // this (object calling isLargerThan)
    // and other must be instances of
    // the same class returns 1, 0, -1
    // if this is greater than,
    // equal to, or less than other
    public int isLargerThan(Relatable other);
}
```

If you want to be able to compare the size of similar objects, no matter what they are, the class that instantiates them should implement `Relatable`.

Any class can implement `Relatable` if there is some way to compare the relative "size" of objects instantiated from the class. For strings, it could be number of characters; for books, it could be number of pages; for students, it could be weight; and so forth. For planar geometric objects, area would be a good choice (see the `RectanglePlus` class that follows), while volume would work for three-dimensional geometric objects. All such classes can implement the `isLargerThan()` method.

If you know that a class implements `Relatable`, then you know that you can compare the size of the objects instantiated from that class.

Implementing the Relatable Interface

Here is the `Rectangle` class that was presented in the Creating Objects section, rewritten to implement `Relatable`.

```
public class RectanglePlus
    implements Relatable {
    public int width = 0;
    public int height = 0;
    public Point origin;

    // four constructors
    public RectanglePlus() {
        origin = new Point(0, 0);
    }
    public RectanglePlus(Point p) {
        origin = p;
    }
    public RectanglePlus(int w, int h) {
        origin = new Point(0, 0);
        width = w;
        height = h;
    }
    public RectanglePlus(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }

    // a method for moving the rectangle
    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }

    // a method for computing
    // the area of the rectangle
    public int getArea() {
        return width * height;
    }

    // a method required to implement
    // the Relatable interface
    public int isLargerThan(Relatable other) {
        RectanglePlus otherRect
            = (RectanglePlus)other;
        if (this.getArea() < otherRect.getArea())
            return -1;
        else if (this.getArea() > otherRect.getArea())
            return 1;
        else
            return 0;
    }
}
```

Because `RectanglePlus` implements `Relatable`, the size of any two `RectanglePlus` objects can be compared.

Note: The `isLargerThan` method, as defined in the `Relatable` interface, takes an object of type `Relatable`. The line of code, shown in bold in the previous example, casts `other` to a `RectanglePlus` instance. Type casting tells the compiler what the object really is. Invoking `getArea` directly on the `other` instance (`other.getArea()`) would fail to compile because the compiler does not understand that `other` is actually an instance of `RectanglePlus`.

Using an Interface as a Type

When you define a new interface, you are defining a new reference data type. You can use interface names anywhere you can use any other data type name. If you define a reference variable whose type is an interface, any object you assign to it *must* be an instance of a class that implements the interface.

As an example, here is a method for finding the largest object in a pair of objects, for *any* objects that are instantiated from a class that implements `Relatable`:

```
public Object findLargest(Object object1, Object object2) {
   Relatable obj1 = (Relatable)object1;
   Relatable obj2 = (Relatable)object2;
   if ((obj1).isLargerThan(obj2) > 0)
      return object1;
   else
      return object2;
}
```

By casting `object1` to a `Relatable` type, it can invoke the `isLargerThan` method.

If you make a point of implementing `Relatable` in a wide variety of classes, the objects instantiated from *any* of those classes can be compared with the `findLargest()` method—provided that both objects are of the same class. Similarly, they can all be compared with the following methods:

```
public Object findSmallest(Object object1, Object object2) {
   Relatable obj1 = (Relatable)object1;
   Relatable obj2 = (Relatable)object2;
   if ((obj1).isLargerThan(obj2) < 0)
      return object1;
   else
      return object2;
}

public boolean isEqual(Object object1, Object object2) {
   Relatable obj1 = (Relatable)object1;
   Relatable obj2 = (Relatable)object2;
   if ( (obj1).isLargerThan(obj2) == 0)
      return true;
   else
      return false;
}
```

These methods work for any "relatable" objects, no matter what their class inheritance is. When they implement `Relatable`, they can be of both their own class (or superclass) type and a `Relatable` type. This gives them some of the advantages of multiple inheritance, where they can have behavior from both a superclass and an interface.

Evolving Interfaces

Consider an interface that you have developed called `DoIt`:

```
public interface DoIt {
   void doSomething(int i, double x);
   int doSomethingElse(String s);
}
```

Suppose that, at a later time, you want to add a third method to `DoIt`, so that the interface now becomes:

```
public interface DoIt {

   void doSomething(int i, double x);
   int doSomethingElse(String s);
   boolean didItWork(int i, double x, String s);

}
```

If you make this change, then all classes that implement the old `DoIt` interface will break because they no longer implement the old interface. Programmers relying on this interface will protest loudly.

Try to anticipate all uses for your interface and specify it completely from the beginning. If you want to add additional methods to an interface, you have several options. You could create a `DoItPlus` interface that extends `DoIt`:

```
public interface DoItPlus extends DoIt {

   boolean didItWork(int i, double x, String s);

}
```

Now users of your code can choose to continue to use the old interface or to upgrade to the new interface.

Alternatively, you can define your new methods as default methods. The following example defines a default method named `didItWork`:

```
public interface DoIt {

   void doSomething(int i, double x);
   int doSomethingElse(String s);
   default boolean didItWork(int i, double x, String s) {
       // Method body
   }

}
```

Note that you must provide an implementation for default methods. You could also define new static methods to existing interfaces. Users who have classes that implement interfaces enhanced with new default or static methods do not have to modify or recompile them to accommodate the additional methods.

Default Methods

The section Interfaces describes an example that involves manufacturers of computer-controlled cars who publish industry-standard interfaces that describe which methods can be invoked to operate their cars. What if those computer-controlled car manufacturers add new functionality, such as flight, to their cars? These manufacturers would need to specify new methods to enable other companies (such as electronic guidance instrument manufacturers) to adapt their software to flying cars. Where would these car manufacturers declare these new flight-related methods? If they add them to their original interfaces, then programmers who have implemented those interfaces would have to rewrite their implementations. If they add them as static methods, then programmers would regard them as utility methods, not as essential, core methods.

Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

Consider the following interface, `TimeClient`, as described in Answers to Questions and Exercises: Interfaces:

```
import java.time.*;

public interface TimeClient {
    void setTime(int hour, int minute, int second);
    void setDate(int day, int month, int year);
    void setDateAndTime(int day, int month, int year,
                               int hour, int minute, int second);
    LocalDateTime getLocalDateTime();
}
```

The following class, `SimpleTimeClient`, implements `TimeClient`:

```
package defaultmethods;

import java.time.*;
import java.lang.*;
import java.util.*;

public class SimpleTimeClient implements TimeClient {

    private LocalDateTime dateAndTime;

    public SimpleTimeClient() {
        dateAndTime = LocalDateTime.now();
    }

    public void setTime(int hour, int minute, int second) {
        LocalDate currentDate = LocalDate.from(dateAndTime);
        LocalTime timeToSet = LocalTime.of(hour, minute, second);
        dateAndTime = LocalDateTime.of(currentDate, timeToSet);
    }

    public void setDate(int day, int month, int year) {
        LocalDate dateToSet = LocalDate.of(day, month, year);
        LocalTime currentTime = LocalTime.from(dateAndTime);
        dateAndTime = LocalDateTime.of(dateToSet, currentTime);
    }

    public void setDateAndTime(int day, int month, int year,
                               int hour, int minute, int second) {
        LocalDate dateToSet = LocalDate.of(day, month, year);
        LocalTime timeToSet = LocalTime.of(hour, minute, second);
        dateAndTime = LocalDateTime.of(dateToSet, timeToSet);
    }

    public LocalDateTime getLocalDateTime() {
        return dateAndTime;
    }

    public String toString() {
        return dateAndTime.toString();
    }

    public static void main(String... args) {
        TimeClient myTimeClient = new SimpleTimeClient();
        System.out.println(myTimeClient.toString());
    }
}
```

Suppose that you want to add new functionality to the `TimeClient` interface, such as the ability to specify a time zone through a `ZonedDateTime` object (which is like a `LocalDateTime` object except that it stores time zone information):

```
public interface TimeClient {
    void setTime(int hour, int minute, int second);
    void setDate(int day, int month, int year);
    void setDateAndTime(int day, int month, int year,
        int hour, int minute, int second);
    LocalDateTime getLocalDateTime();
    ZonedDateTime getZonedDateTime(String zoneString);
}
```

Following this modification to the `TimeClient` interface, you would also have to modify the class `SimpleTimeClient` and implement the method `getZonedDateTime`. However, rather than leaving `getZonedDateTime` as `abstract` (as in the previous example), you can instead define a *default implementation*. (Remember that an abstract method is a method declared without an implementation.)

```


package defaultmethods;

import java.time.*;

public interface TimeClient {
    void setTime(int hour, int minute, int second);
    void setDate(int day, int month, int year);
    void setDateAndTime(int day, int month, int year,
                               int hour, int minute, int second);
    LocalDateTime getLocalDateTime();

    static ZoneId getZoneId (String zoneString) {
        try {
            return ZoneId.of(zoneString);
        } catch (DateTimeException e) {
            System.err.println("Invalid time zone: " + zoneString +
                "; using default time zone instead.");
            return ZoneId.systemDefault();
        }
    }

    default ZonedDateTime getZonedDateTime(String zoneString) {
        return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));
    }
}
```

You specify that a method definition in an interface is a default method with the `default` keyword at the beginning of the method signature. All method declarations in an interface, including default methods, are implicitly `public`, so you can omit the `public` modifier.

With this interface, you do not have to modify the class `SimpleTimeClient`, and this class (and any class that implements the interface `TimeClient`), will have the method `getZonedDateTime` already defined. The following example, `TestSimpleTimeClient`, invokes the method `getZonedDateTime` from an instance of `SimpleTimeClient`:

```
package defaultmethods;

import java.time.*;
import java.lang.*;
import java.util.*;

public class TestSimpleTimeClient {
    public static void main(String... args) {
        TimeClient myTimeClient = new SimpleTimeClient();
        System.out.println("Current time: " + myTimeClient.toString());
        System.out.println("Time in California: " +
            myTimeClient.getZonedDateTime("Blah blah").toString());
    }
}
```

Extending Interfaces That Contain Default Methods

When you extend an interface that contains a default method, you can do the following:

- Not mention the default method at all, which lets your extended interface inherit the default method.
- Redeclare the default method, which makes it `abstract`.
- Redefine the default method, which overrides it.

Suppose that you extend the interface `TimeClient` as follows:

```
public interface AnotherTimeClient extends TimeClient { }
```

Any class that implements the interface `AnotherTimeClient` will have the implementation specified by the default method `TimeClient.getZonedDateTime`.

Suppose that you extend the interface `TimeClient` as follows:

```
public interface AbstractZoneTimeClient extends TimeClient {
    public ZonedDateTime getZonedDateTime(String zoneString);
}
```

Any class that implements the interface `AbstractZoneTimeClient` will have to implement the method `getZonedDateTime`; this method is an `abstract` method like all other nondefault (and nonstatic) methods in an interface.

Suppose that you extend the interface `TimeClient` as follows:

```
public interface HandleInvalidTimeZoneClient extends TimeClient {
    default public ZonedDateTime getZonedDateTime(String zoneString) {
        try {
            return ZonedDateTime.of(getLocalDateTime(),ZoneId.of(zoneString));
        } catch (DateTimeException e) {
            System.err.println("Invalid zone ID: " + zoneString +
                "; using the default time zone instead.");
            return ZonedDateTime.of(getLocalDateTime(),ZoneId.systemDefault());
        }
    }
}
```

Any class that implements the interface `HandleInvalidTimeZoneClient` will use the implementation of `getZonedDateTime` specified by this interface instead of the one specified by the interface `TimeClient`.

Static Methods

In addition to default methods, you can define static methods in interfaces. (A static method is a method that is associated with the class in which it is defined rather than with any object. Every instance of the class shares its static methods.) This makes it easier for you to organize helper methods in your libraries; you can keep static methods specific to an interface in the same interface rather than in a separate class. The following example defines a static method that retrieves a `ZoneId` object corresponding to a time zone identifier; it uses the system default time zone if there is no `ZoneId` object corresponding to the given identifier. (As a result, you can simplify the method `getZonedDateTime`):

```
public interface TimeClient {
    // ...
    static public ZoneId getZoneId (String zoneString) {
        try {
            return ZoneId.of(zoneString);
        } catch (DateTimeException e) {
            System.err.println("Invalid time zone: " + zoneString +
                "; using default time zone instead.");
            return ZoneId.systemDefault();
        }
    }

    default public ZonedDateTime getZonedDateTime(String zoneString) {
        return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));
    }
}
```

Like static methods in classes, you specify that a method definition in an interface is a static method with the `static` keyword at the beginning of the method signature. All method declarations in an interface, including static methods, are implicitly `public`, so you can omit the `public` modifier.

Integrating Default Methods into Existing Libraries

Default methods enable you to add new functionality to existing interfaces and ensure binary compatibility with code written for older versions of those interfaces. In particular, default methods enable you to add methods that accept lambda expressions as parameters to existing interfaces. This section demonstrates how the `Comparator` interface has been enhanced with default and static methods.

Consider the `Card` and `Deck` classes as described in Questions and Exercises: Classes. This example rewrites the `Card` and `Deck` classes as interfaces. The `Card` interface contains two `enum` types (`Suit` and `Rank`) and two abstract methods (`getSuit` and `getRank`):

```
package defaultmethods;

public interface Card extends Comparable<Card> {

    public enum Suit {
        DIAMONDS (1, "Diamonds"),
        CLUBS    (2, "Clubs"   ),
        HEARTS   (3, "Hearts"  ),
        SPADES   (4, "Spades"  );

        private final int value;
        private final String text;
        Suit(int value, String text) {
            this.value = value;
            this.text = text;
        }
        public int value() {return value;}
        public String text() {return text;}
    }

    public enum Rank {
        DEUCE  (2 , "Two"  ),
        THREE  (3 , "Three"),
        FOUR   (4 , "Four" ),
        FIVE   (5 , "Five" ),
        SIX    (6 , "Six"  ),
        SEVEN  (7 , "Seven"),
        EIGHT  (8 , "Eight"),
        NINE   (9 , "Nine" ),
        TEN    (10, "Ten"  ),
        JACK   (11, "Jack" ),
        QUEEN  (12, "Queen"),
        KING   (13, "King" ),
        ACE    (14, "Ace"  );
        private final int value;
        private final String text;
        Rank(int value, String text) {
            this.value = value;
            this.text = text;
        }
        public int value() {return value;}
        public String text() {return text;}
    }
    public Card.Suit getSuit();
    public Card.Rank getRank();
}
```

The `Deck` interface contains various methods that manipulate cards in a deck:

```
package defaultmethods;

import java.util.*;
import java.util.stream.*;
import java.lang.*;

public interface Deck {

    List<Card> getCards();
    Deck deckFactory();
    int size();
    void addCard(Card card);
    void addCards(List<Card> cards);
    void addDeck(Deck deck);
    void shuffle();
    void sort();
    void sort(Comparator<Card> c);
    String deckToString();

    Map<Integer, Deck> deal(int players, int numberOfCards)
        throws IllegalArgumentException;

}
```

The class `PlayingCard` implements the interface `Card`, and the class `StandardDeck` implements the interface `Deck`.

The class `StandardDeck` implements the abstract method `Deck.sort` as follows:

```
public class StandardDeck implements Deck {

    private List<Card> entireDeck;

    // ...

    public void sort() {
        Collections.sort(entireDeck);
    }

    // ...
}
```

The method `Collections.sort` sorts an instance of `List` whose element type implements the interface `Comparable`. The member `entireDeck` is an instance of `List` whose elements are of the type `Card`, which extends `Comparable`. The class `PlayingCard` implements the `Comparable.compareTo` method as follows:

```
public int hashCode() {
    return ((suit.value()-1)*13)+rank.value();
}

public int compareTo(Card o) {
    return this.hashCode() - o.hashCode();
}
```

The method `compareTo` causes the method `StandardDeck.sort()` to sort the deck of cards first by suit, and then by rank.

What if you want to sort the deck first by rank, then by suit? You would need to implement the `Comparator` interface to specify new sorting criteria, and use the method `sort(List<T> list, Comparator<? super T> c)` (the version of the `sort` method that includes a `Comparator` parameter). You can define the following method in the class `StandardDeck`:

```
public void sort(Comparator<Card> c) {
    Collections.sort(entireDeck, c);
}
```

With this method, you can specify how the method `Collections.sort` sorts instances of the `Card` class. One way to do this is to implement the `Comparator` interface to specify how you want the cards sorted. The example `SortByRankThenSuit` does this:

```
package defaultmethods;

import java.util.*;
import java.util.stream.*;
import java.lang.*;

public class SortByRankThenSuit implements Comparator<Card> {
    public int compare(Card firstCard, Card secondCard) {
        int compVal =
            firstCard.getRank().value() - secondCard.getRank().value();
        if (compVal != 0)
            return compVal;
        else
            return firstCard.getSuit().value() - secondCard.getSuit().value();
    }
}
```

The following invocation sorts the deck of playing cards first by rank, then by suit:

```
StandardDeck myDeck = new StandardDeck();
myDeck.shuffle();
myDeck.sort(new SortByRankThenSuit());
```

However, this approach is too verbose; it would be better if you could specify *what* you want to sort, not *how* you want to sort. Suppose that you are the developer who wrote the `Comparator` interface. What default or static methods could you add to the `Comparator` interface to enable other developers to more easily specify sort criteria?

To start, suppose that you want to sort the deck of playing cards by rank, regardless of suit. You can invoke the `StandardDeck.sort` method as follows:

```
StandardDeck myDeck = new StandardDeck();
myDeck.shuffle();
myDeck.sort(
    (firstCard, secondCard) ->
        firstCard.getRank().value() - secondCard.getRank().value()
);
```

Because the interface `Comparator` is a functional interface, you can use a lambda expression as an argument for the `sort` method. In this example, the lambda expression compares two integer values.

It would be simpler for your developers if they could create a `Comparator` instance by invoking the method `Card.getRank` only. In particular, it would be helpful if your developers could create a `Comparator` instance that compares any object that can return a numerical value from a method such as `getValue` or `hashCode`. The `Comparator` interface has been enhanced with this ability with the static method `comparing`:

```
myDeck.sort(Comparator.comparing((card) -> card.getRank()));
```

In this example, you can use a method reference instead:

```
myDeck.sort(Comparator.comparing(Card::getRank()));
```

This invocation better demonstrates *what* to sort rather than *how* to do it.

The `Comparator` interface has been enhanced with other versions of the static method `comparing` such as `comparingDouble` and `comparingLong` that enable you to create `Comparator` instances that compare other data types.

Suppose that your developers would like to create a `Comparator` instance that could compare objects with more than one criteria. For example, how would you sort the deck of playing cards first by rank, and then by suit? As before, you could use a lambda expression to specify these sort criteria:

```
StandardDeck myDeck = new StandardDeck();
myDeck.shuffle();
myDeck.sort(
    (firstCard, secondCard) -> {
        int compare =
            firstCard.getRank().value() - secondCard.getRank().value();
        if (compare != 0)
            return compare;
        else
            return firstCard.getSuit().value() - secondCard.getSuit().value();
    }
);
```

It would be simpler for your developers if they could build a `Comparator` instance from a series of `Comparator` instances. The `Comparator` interface has been enhanced with this ability with the default method `thenComparing`:

```
myDeck.sort(
    Comparator
        .comparing(Card::getRank)
        .thenComparing(Comparator.comparing(Card::getSuit)));
```

The `Comparator` interface has been enhanced with other versions of the default method `thenComparing` (such as `thenComparingDouble` and `thenComparingLong`) that enable you to build `Comparator` instances that compare other data types.

Suppose that your developers would like to create a `Comparator` instance that enables them to sort a collection of objects in reverse order. For example, how would you sort the deck of playing cards first by descending order of rank, from Ace to Two (instead of from Two to Ace)? As before, you could specify another lambda expression. However, it would be simpler for your developers if they could reverse an existing `Comparator` by invoking a method. The `Comparator` interface has been enhanced with this ability with the default method `reversed`:

```
myDeck.sort(
    Comparator.comparing(Card::getRank)
        .reversed()
        .thenComparing(Comparator.comparing(Card::getSuit)));
```

This example demonstrates how the `Comparator` interface has been enhanced with default methods, static methods, lambda expressions, and method references to create more expressive library methods whose functionality programmers can quickly deduce by looking at how they are invoked. Use these constructs to enhance the interfaces in your libraries.

Summary of Interfaces

An interface declaration can contain method signatures, default methods, static methods and constant definitions. The only methods that have implementations are default and static methods.

A class that implements an interface must implement all the methods declared in the interface.

An interface name can be used anywhere a type can be used.

Questions and Exercises: Interfaces

Questions

1. What methods would a class that implements the `java.lang.CharSequence` interface have to implement?

2. What is wrong with the following interface?

```
public interface SomethingIsWrong {
    void aMethod(int aValue){
        System.out.println("Hi Mom");
    }
}
```

3. Fix the interface in question 2.

4. Is the following interface valid?

```
public interface Marker {
}
```

Exercises

1. Write a class that implements the `CharSequence` interface found in the `java.lang` package. Your implementation should return the string backwards. Select one of the sentences from this book to use as the data. Write a small `main` method to test your class; make sure to call all four methods.

2. Suppose you have written a time server that periodically notifies its clients of the current date and time. Write an interface the server could use to enforce a particular protocol on its clients.

Check your answers.

Questions

Question 1: What methods would a class that implements the `java.lang.CharSequence` interface have to implement?
Answer 1: `charAt`, `length`, `subSequence`, and `toString`.

Question 2: What is wrong with the following interface?

```
public interface SomethingIsWrong {
    void aMethod(int aValue) {
        System.out.println("Hi Mom");
    }
}
```

Answer 2: It has a method implementation in it. Only default and static methods have implementations.

Question 3: Fix the interface in Question 2.
Answer 3:

```
public interface SomethingIsWrong {
    void aMethod(int aValue);
}
```

Alternatively, you can define `aMethod` as a default method:

```
public interface SomethingIsWrong {
    default void aMethod(int aValue) {
        System.out.println("Hi Mom");
    }
}
```

Question 4: Is the following interface valid?

```
public interface Marker {
}
```

Answer 4: Yes. Methods are not required. Empty interfaces can be used as types and to mark classes without requiring any particular method implementations. For an example of a useful empty interface, see `java.io.Serializable`.

Exercises

Exercise 1: Write a class that implements the `CharSequence` interface found in the `java.lang` package. Your implementation should return the string backwards. Select one of the sentences from this book to use as the data. Write a small `main` method to test your class; make sure to call all four methods.

Answer 1: See `CharSequenceDemo.java`

Exercise 2: Suppose that you have written a time server, which periodically notifies its clients of the current date and time. Write an interface that the server could use to enforce a particular protocol on its clients.
Answer 2: See `TimeClient.java`.

Inheritance

In the preceding lessons, you have seen *inheritance* mentioned several times. In the Java language, classes can be *derived* from other classes, thereby *inheriting* fields and methods from those classes.

Definitions: A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*). The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*).

Excepting `Object`, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of `Object`.

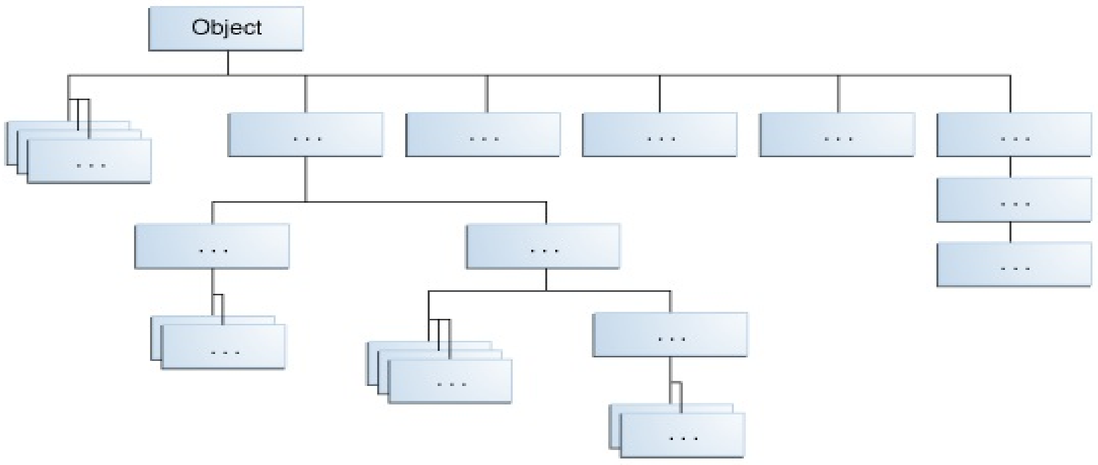
Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, `Object`. Such a class is said to be *descended* from all the classes in the inheritance chain stretching back to `Object`.

The idea of inheritance is simple but powerful: When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself.

A subclass inherits all the *members* (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

The Java Platform Class Hierarchy

The `Object` class, defined in the `java.lang` package, defines and implements behavior common to all classes—including the ones that you write. In the Java platform, many classes derive directly from `Object`, other classes derive from some of those classes, and so on, forming a hierarchy of classes.



All Classes in the Java Platform are Descendants of Object

At the top of the hierarchy, `Object` is the most general of all classes. Classes near the bottom of the hierarchy provide more specialized behavior.

An Example of Inheritance

Here is the sample code for a possible implementation of a `Bicycle` class that was presented in the Classes and Objects lesson:

```
public class Bicycle {

    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

}
```

A class declaration for a `MountainBike` class that is a subclass of `Bicycle` might look like this:

```
public class MountainBike extends Bicycle {

    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight,
                        int startCadence,
                        int startSpeed,
                        int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

`MountainBike` inherits all the fields and methods of `Bicycle` and adds the field `seatHeight` and a method to set it. Except for the constructor, it is as if you had written a new `MountainBike` class entirely from scratch, with four fields and five methods. However, you didn't have to do all the work. This would be especially valuable if the methods in the `Bicycle` class were complex and had taken substantial time to debug.

What You Can Do in a Subclass

A subclass inherits all of the *public* and *protected* members of its parent, no matter what package the subclass is in. If the subclass is in the same package as its parent, it also inherits the *package-private* members of the parent. You can use the inherited members as is, replace them, hide them, or supplement them with new members:

- The inherited fields can be used directly, just like any other fields.
- You can declare a field in the subclass with the same name as the one in the superclass, thus *hiding* it (not recommended).
- You can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *overriding* it.
- You can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus *hiding* it.
- You can declare new methods in the subclass that are not in the superclass.
- You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword `super`.

The following sections in this lesson will expand on these topics.

Private Members in a Superclass

A subclass does not inherit the `private` members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

A nested class has access to all the private members of its enclosing class—both fields and methods. Therefore, a public or protected nested class inherited by a subclass has indirect access to all of the private members of the superclass.

Casting Objects

We have seen that an object is of the data type of the class from which it was instantiated. For example, if we write

```
public MountainBike myBike = new MountainBike();
```

then `myBike` is of type `MountainBike`.

`MountainBike` is descended from `Bicycle` and `Object`. Therefore, a `MountainBike` is a `Bicycle` and is also an `Object`, and it can be used wherever `Bicycle` or `Object` objects are called for.

The reverse is not necessarily true: a `Bicycle` *may be* a `MountainBike`, but it isn't necessarily. Similarly, an `Object` *may be* a `Bicycle` or a `MountainBike`, but it isn't necessarily.

Casting shows the use of an object of one type in place of another type, among the objects permitted by inheritance and implementations. For example, if we write

```
Object obj = new MountainBike();
```

then `obj` is both an `Object` and a `MountainBike` (until such time as `obj` is assigned another object that is *not* a `MountainBike`). This is called *implicit casting*.

If, on the other hand, we write

```
MountainBike myBike = obj;
```

we would get a compile-time error because `obj` is not known to the compiler to be a `MountainBike`. However, we can *tell* the compiler that we promise to assign a `MountainBike` to `obj` by *explicit casting:*

```
MountainBike myBike = (MountainBike)obj;
```

This cast inserts a runtime check that `obj` is assigned a `MountainBike` so that the compiler can safely assume that `obj` is a `MountainBike`. If `obj` is not a `MountainBike` at runtime, an exception will be thrown.

Note: You can make a logical test as to the type of a particular object using the `instanceof` operator. This can save you from a runtime error owing to an improper cast. For example:

```
if (obj instanceof MountainBike) {
    MountainBike myBike = (MountainBike)obj;
}
```

Here the `instanceof` operator verifies that `obj` refers to a `MountainBike` so that we can make the cast with knowledge that there will be no runtime exception thrown.

Multiple Inheritance of State, Implementation, and Type

One significant difference between classes and interfaces is that classes can have fields whereas interfaces cannot. In addition, you can instantiate a class to create an object, which you cannot do with interfaces. As explained in the section What Is an Object?, an object stores its state in fields, which are defined in classes. One reason why the Java programming language does not permit you to extend more than one class is to avoid the issues of *multiple inheritance of state*, which is the ability to inherit fields from multiple classes. For example, suppose that you are able to define a new class that extends multiple classes. When you create an object by instantiating that class, that object will inherit fields from all of the class's superclasses. What if methods or constructors from different superclasses instantiate the same field? Which method or constructor will take precedence? Because interfaces do not contain fields, you do not have to worry about problems that result from multiple inheritance of state.

Multiple inheritance of implementation is the ability to inherit method definitions from multiple classes. Problems arise with this type of multiple inheritance, such as name conflicts and ambiguity. When compilers of programming languages that support this type of multiple inheritance encounter superclasses that contain methods with the same name, they sometimes cannot determine which member or method to access or invoke. In addition, a programmer can unwittingly introduce a name conflict by adding a new method to a superclass. Default methods introduce one form of multiple inheritance of implementation. A class can implement more than one interface, which can contain default methods that have the same name. The Java compiler provides some rules to determine which default method a particular class uses.

The Java programming language supports *multiple inheritance of type*, which is the ability of a class to implement more than one interface. An object can have multiple types: the type of its own class and the types of all the interfaces that the class implements. This means that if a variable is declared to be the type of an interface, then its value can reference any object that is instantiated from any class that implements the interface. This is discussed in the section Using an Interface as a Type.

As with multiple inheritance of implementation, a class can inherit different implementations of a method defined (as default or static) in the interfaces that it extends. In this case, the compiler or the user must decide which one to use.

Overriding and Hiding Methods

Instance Methods

An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass *overrides* the superclass's method.

The ability of a subclass to override a method allows a class to inherit from a superclass whose behavior is "close enough" and then to modify behavior as needed. The overriding method has the same name, number and type of parameters, and return type as the method that it overrides. An overriding method can also return a subtype of the type returned by the overridden method. This subtype is called a *covariant return type*.

When overriding a method, you might want to use the `@Override` annotation that instructs the compiler that you intend to override a method in the superclass. If, for some reason, the compiler detects that the method does not exist in one of the superclasses, then it will generate an error. For more information on `@Override`, see Annotations.

Static Methods

If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass *hides* the one in the superclass.

The distinction between hiding a static method and overriding an instance method has important implications:

- The version of the overridden instance method that gets invoked is the one in the subclass.
- The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.

Consider an example that contains two classes. The first is `Animal`, which contains one instance method and one static method:

```
public class Animal {
    public static void testClassMethod() {
        System.out.println("The static method in Animal");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal");
    }
}
```

The second class, a subclass of `Animal`, is called `Cat`:

```
public class Cat extends Animal {
    public static void testClassMethod() {
        System.out.println("The static method in Cat");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat");
    }

    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = myCat;
        Animal.testClassMethod();
        myAnimal.testInstanceMethod();
    }
}
```

The `Cat` class overrides the instance method in `Animal` and hides the static method in `Animal`. The `main` method in this class creates an instance of `Cat` and invokes `testClassMethod()` on the class and `testInstanceMethod()` on the instance.

The output from this program is as follows:

```
The static method in Animal
The instance method in Cat
```

As promised, the version of the hidden static method that gets invoked is the one in the superclass, and the version of the overridden instance method that gets invoked is the one in the subclass.

Interface Methods

Default methods and abstract methods in interfaces are inherited like instance methods. However, when the supertypes of a class or interface provide multiple default methods with the same signature, the Java compiler follows inheritance rules to resolve the name conflict. These rules are driven by the following two principles:

- Instance methods are preferred over interface default methods.
  Consider the following classes and interfaces:
  ```
  public class Horse {
      public String identifyMyself() {
          return "I am a horse.";
      }
  }
  ```
  ```
  public interface Flyer {
      default public String identifyMyself() {
          return "I am able to fly.";
      }
  }
  ```
  ```
  public interface Mythical {
      default public String identifyMyself() {
          return "I am a mythical creature.";
      }
  }
  ```
  ```
  public class Pegasus extends Horse implements Flyer, Mythical {
      public static void main(String... args) {
          Pegasus myApp = new Pegasus();
          System.out.println(myApp.identifyMyself());
      }
  }
  ```
  The method `Pegasus.identifyMyself` returns the string `I am a horse.`
- Methods that are already overridden by other candidates are ignored. This circumstance can

arise when supertypes share a common ancestor.
Consider the following interfaces and classes:

```
public interface Animal {
    default public String identifyMyself() {
        return "I am an animal.";
    }
}
```

```
public interface EggLayer extends Animal {
    default public String identifyMyself() {
        return "I am able to lay eggs.";
    }
}
```

```
public interface FireBreather extends Animal { }
```

```
public class Dragon implements EggLayer, FireBreather {
    public static void main (String... args) {
        Dragon myApp = new Dragon();
        System.out.println(myApp.identifyMyself());
    }
}
```

The method `Dragon.identifyMyself` returns the string `I am able to lay eggs.`

If two or more independently defined default methods conflict, or a default method conflicts with an abstract method, then the Java compiler produces a compiler error. You must explicitly override the supertype methods.

Consider the example about computer-controlled cars that can now fly. You have two interfaces (`OperateCar` and `FlyCar`) that provide default implementations for the same method, (`startEngine`):

```
public interface OperateCar {
    // ...
    default public int startEngine(EncryptedKey key) {
        // Implementation
    }
}
```

```
public interface FlyCar {
    // ...
    default public int startEngine(EncryptedKey key) {
        // Implementation
    }
}
```

A class that implements both `OperateCar` and `FlyCar` must override the method `startEngine`. You could invoke any of the of the default implementations with the `super` keyword.

```
public class FlyingCar implements OperateCar, FlyCar {
    // ...
    public int startEngine(EncryptedKey key) {
        FlyCar.super.startEngine(key);
        OperateCar.super.startEngine(key);
    }
}
```

The name preceding `super` (in this example, `FlyCar` or `OperateCar`) must refer to a direct superinterface that defines or inherits a default for the invoked method. This form of method invocation is not restricted to differentiating between multiple implemented interfaces that contain default methods with the same signature. You can use the `super` keyword to invoke a default method in both classes and interfaces.

Inherited instance methods from classes can override abstract interface methods. Consider the following interfaces and classes:

```
public interface Mammal {
    String identifyMyself();
}
```

```
public class Horse {
    public String identifyMyself() {
        return "I am a horse.";
    }
}
```

```
public class Mustang extends Horse implements Mammal {
    public static void main(String... args) {
        Mustang myApp = new Mustang();
        System.out.println(myApp.identifyMyself());
    }
}
```

The method `Mustang.identifyMyself` returns the string `I am a horse.` The class `Mustang` inherits the method `identifyMyself` from the class `Horse`, which overrides the abstract method of the same name in the interface `Mammal`.

Note: Static methods in interfaces are never inherited.

Modifiers

The access specifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the superclass can be made public, but not private, in the subclass.

You will get a compile-time error if you attempt to change an instance method in the superclass to a static method in the subclass, and vice versa.

Summary

The following table summarizes what happens when you define a method with the same signature as a method in a superclass.

Defining a Method with the Same Signature as a Superclass's Method

	Superclass Instance Method	**Superclass Static Method**
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

Note: In a subclass, you can overload the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass instance methods—they are new methods, unique to the subclass.

Polymorphism

The dictionary definition of *polymorphism* refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming and languages like the Java language. Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

Polymorphism can be demonstrated with a minor modification to the `Bicycle` class. For example, a `printDescription` method could be added to the class that displays all the data currently stored in an instance.

```
public void printDescription(){
    System.out.println("\nBike is " + "in gear " + this.gear
        + " with a cadence of " + this.cadence +
        " and travelling at a speed of " + this.speed + ". ");
}
```

To demonstrate polymorphic features in the Java language, extend the `Bicycle` class with a `MountainBike` and a `RoadBike` class. For `MountainBike`, add a field for `suspension`, which is a `String` value that indicates if the bike has a front shock absorber, `Front`. Or, the bike has a front and back shock absorber, `Dual`.

Here is the updated class:

```
public class MountainBike extends Bicycle {
    private String suspension;

    public MountainBike(
               int startCadence,
               int startSpeed,
               int startGear,
               String suspensionType){
        super(startCadence,
              startSpeed,
              startGear);
        this.setSuspension(suspensionType);
    }

    public String getSuspension(){
      return this.suspension;
    }

    public void setSuspension(String suspensionType) {
        this.suspension = suspensionType;
    }

    public void printDescription() {
        super.printDescription();
        System.out.println("The " + "MountainBike has a" +
            getSuspension() + " suspension.");
    }
}
```

Note the overridden `printDescription` method. In addition to the information provided before, additional data about the suspension is included to the output.

Next, create the RoadBike class. Because road or racing bikes have skinny tires, add an attribute to track the tire width. Here is the RoadBike class:

```
public class RoadBike extends Bicycle{
    // In millimeters (mm)
    private int tireWidth;

    public RoadBike(int startCadence,
                    int startSpeed,
                    int startGear,
                    int newTireWidth){
        super(startCadence,
              startSpeed,
              startGear);
        this.setTireWidth(newTireWidth);
    }

    public int getTireWidth(){
      return this.tireWidth;
    }

    public void setTireWidth(int newTireWidth){
        this.tireWidth = newTireWidth;
    }

    public void printDescription(){
        super.printDescription();
        System.out.println("The RoadBike" + " has " + getTireWidth() +
            " MM tires.");
    }
}
```

Note that once again, the printDescription method has been overridden. This time, information about the tire width is displayed.

To summarize, there are three classes: Bicycle, MountainBike, and RoadBike. The two subclasses override the printDescription method and print unique information.

Here is a test program that creates three Bicycle variables. Each variable is assigned to one of the three bicycle classes. Each variable is then printed.

```
public class TestBikes {
  public static void main(String[] args){
    Bicycle bike01, bike02, bike03;

    bike01 = new Bicycle(20, 10, 1);
    bike02 = new MountainBike(20, 10, 5, "Dual");
    bike03 = new RoadBike(40, 20, 8, 23);

    bike01.printDescription();
    bike02.printDescription();
    bike03.printDescription();
  }
}
```

The following is the output from the test program:

```
Bike is in gear 1 with a cadence of 20 and travelling at a speed of 10.
Bike is in gear 5 with a cadence of 20 and travelling at a speed of 10.
The MountainBike has a Dual suspension.

Bike is in gear 8 with a cadence of 40 and travelling at a speed of 20.
The RoadBike has 23 MM tires.
```

The Java virtual machine (JVM) calls the appropriate method for the object that is referred to in each variable. It does not call the method that is defined by the variable's type. This behavior is referred to as *virtual method invocation* and demonstrates an aspect of the important polymorphism features in the Java language.

Hiding Fields

Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different. Within the subclass, the field in the superclass cannot be referenced by its simple name. Instead, the field must be accessed through `super`, which is covered in the next section. Generally speaking, we don't recommend hiding fields as it makes code difficult to read.

Using the Keyword super

Accessing Superclass Members

If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword `super`. You can also use `super` to refer to a hidden field (although hiding fields is discouraged). Consider this class, `Superclass`:

```
public class Superclass {

    public void printMethod() {
        System.out.println("Printed in Superclass.");
    }
}
```

Here is a subclass, called `Subclass`, that overrides `printMethod()`:

```
public class Subclass extends Superclass {

    // overrides printMethod in Superclass
    public void printMethod() {
        super.printMethod();
        System.out.println("Printed in Subclass");
    }
    public static void main(String[] args) {
        Subclass s = new Subclass();
        s.printMethod();
    }
}
```

Within `Subclass`, the simple name `printMethod()` refers to the one declared in `Subclass`, which overrides the one in `Superclass`. So, to refer to `printMethod()` inherited from `Superclass`, `Subclass` must use a qualified name, using `super` as shown. Compiling and executing `Subclass` prints the following:

```
Printed in Superclass.
Printed in Subclass
```

Subclass Constructors

The following example illustrates how to use the `super` keyword to invoke a superclass's constructor. Recall from the `Bicycle` example that `MountainBike` is a subclass of `Bicycle`. Here is the `MountainBike` (subclass) constructor that calls the superclass constructor and then adds initialization code of its own:

```
public MountainBike(int startHeight,
                    int startCadence,
                    int startSpeed,
                    int startGear) {
    super(startCadence, startSpeed, startGear);
    seatHeight = startHeight;
}
```

Invocation of a superclass constructor must be the first line in the subclass constructor.

The syntax for calling a superclass constructor is

```
super();
```

or:

```
super(parameter list);
```

With `super()`, the superclass no-argument constructor is called. With `super(parameter list)`, the superclass constructor with a matching parameter list is called.

Note: If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the super class does not have a no-argument constructor, you will get a compile-time error. `Object` *does* have such a constructor, so if `Object` is the only superclass, there is no problem.

If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that there will be a whole chain of constructors called, all the way back to the constructor of `Object`. In fact, this is the case. It is called *constructor chaining*, and you need to be aware of it when there is a long line of class descent.

Object as a Superclass

The `Object` class, in the `java.lang` package, sits at the top of the class hierarchy tree. Every class is a descendant, direct or indirect, of the `Object` class. Every class you use or write inherits the instance methods of `Object`. You need not use any of these methods, but, if you choose to do so, you may need to override them with code that is specific to your class. The methods inherited from `Object` that are discussed in this section are:

- `protected Object clone() throws CloneNotSupportedException`
  Creates and returns a copy of this object.
- `public boolean equals(Object obj)`
  Indicates whether some other object is "equal to" this one.
- `protected void finalize() throws Throwable`
  Called by the garbage collector on an object when garbage
  collection determines that there are no more references to the object
- `public final Class getClass()`
  Returns the runtime class of an object.
- `public int hashCode()`
  Returns a hash code value for the object.
- `public String toString()`
  Returns a string representation of the object.

The `notify`, `notifyAll`, and `wait` methods of `Object` all play a part in synchronizing the activities of independently running threads in a program, which is discussed in a later lesson and won't be covered here. There are five of these methods:

- `public final void notify()`
- `public final void notifyAll()`
- `public final void wait()`
- `public final void wait(long timeout)`
- `public final void wait(long timeout, int nanos)`

Note: There are some subtle aspects to a number of these methods, especially the `clone` method.

The clone() Method

If a class, or one of its superclasses, implements the `Cloneable` interface, you can use the `clone()` method to create a copy from an existing object. To create a clone, you write:

```
aCloneableObject.clone();
```

`Object`'s implementation of this method checks to see whether the object on which `clone()` was invoked implements the `Cloneable` interface. If the object does not, the method throws a `CloneNotSupportedException` exception. Exception handling will be covered in a later lesson. For the moment, you need to know that `clone()` must be declared as

```
protected Object clone() throws CloneNotSupportedException
```

or:

```
public Object clone() throws CloneNotSupportedException
```

if you are going to write a `clone()` method to override the one in `Object`.

If the object on which `clone()` was invoked does implement the `Cloneable` interface, `Object`'s implementation of the `clone()` method creates an object of the same class as the original object and initializes the new object's member variables to have the same values as the original object's corresponding member variables.

The simplest way to make your class cloneable is to add `implements Cloneable` to your class's declaration. then your objects can invoke the `clone()` method.

For some classes, the default behavior of `Object`'s `clone()` method works just fine. If, however, an object contains a reference to an external object, say `ObjExternal`, you may need to override `clone()` to get correct behavior. Otherwise, a change in `ObjExternal` made by one object will be visible in its clone also. This means that the original object and its clone are not independent—to decouple them, you must override `clone()` so that it clones the object *and* `ObjExternal`. Then the original object references `ObjExternal` and the clone references a clone of `ObjExternal`, so that the object and its clone are truly independent.

The equals() Method

The `equals()` method compares two objects for equality and returns `true` if they are equal. The `equals()` method provided in the `Object` class uses the identity operator (==) to determine whether two objects are equal. For primitive data types, this gives the correct result. For objects, however, it does not. The `equals()` method provided by `Object` tests whether the object *references* are equal—that is, if the objects compared are the exact same object.

To test whether two objects are equal in the sense of *equivalency* (containing the same information), you must override the `equals()` method. Here is an example of a `Book` class that overrides `equals()`:

```
public class Book {
    ...
    public boolean equals(Object obj) {
        if (obj instanceof Book)
            return ISBN.equals((Book)obj.getISBN());
        else
            return false;
    }
}
```

Consider this code that tests two instances of the `Book` class for equality:

```
// Swing Tutorial, 2nd edition
Book firstBook  = new Book("0201914670");
Book secondBook = new Book("0201914670");
if (firstBook.equals(secondBook)) {
    System.out.println("objects are equal");
} else {
    System.out.println("objects are not equal");
}
```

This program displays `objects are equal` even though `firstBook` and `secondBook` reference two distinct objects. They are considered equal because the objects compared contain the same ISBN number.

You should always override the `equals()` method if the identity operator is not appropriate for your class.

Note: If you override `equals()`, you must override `hashCode()` as well.

The finalize() Method

The `Object` class provides a callback method, `finalize()`, that *may be* invoked on an object when it becomes garbage. `Object`'s implementation of `finalize()` does nothing—you can override `finalize()` to do cleanup, such as freeing resources.

The `finalize()` method *may be* called automatically by the system, but when it is called, or even if it is called, is uncertain. Therefore, you should not rely on this method to do your cleanup for you. For example, if you don't close file descriptors in your code after performing I/O and you expect `finalize()` to close them for you, you may run out of file descriptors.

The getClass() Method

You cannot override `getClass`.

The `getClass()` method returns a `Class` object, which has methods you can use to get information about the class, such as its name (`getSimpleName()`), its superclass (`getSuperclass()`), and the interfaces it implements (`getInterfaces()`). For example, the following method gets and displays the class name of an object:

```
void printClassName(Object obj) {
    System.out.println("The object's" + " class is " +
        obj.getClass().getSimpleName());
}
```

The `Class` class, in the `java.lang` package, has a large number of methods (more than 50). For example, you can test to see if the class is an annotation (`isAnnotation()`), an interface (`isInterface()`), or an enumeration (`isEnum()`). You can see what the object's fields are (`getFields()`) or what its methods are (`getMethods()`), and so on.

The hashCode() Method

The value returned by `hashCode()` is the object's hash code, which is the object's memory address in hexadecimal.

By definition, if two objects are equal, their hash code *must also* be equal. If you override the `equals()` method, you change the way two objects are equated and `Object`'s implementation of `hashCode()` is no longer valid. Therefore, if you override the `equals()` method, you must also override the `hashCode()` method as well.

The toString() Method

You should always consider overriding the `toString()` method in your classes.

The `Object`'s `toString()` method returns a `String` representation of the object, which is very useful for debugging. The `String` representation for an object depends entirely on the object, which is why you need to override `toString()` in your classes.

You can use `toString()` along with `System.out.println()` to display a text representation of an object, such as an instance of `Book`:

```
System.out.println(firstBook.toString());
```

which would, for a properly overridden `toString()` method, print something useful, like this:

```
ISBN: 0201914670; The Swing Tutorial; A Guide to Constructing GUIs, 2nd Edition
```

Writing Final Classes and Methods

You can declare some or all of a class's methods *final*. You use the `final` keyword in a method declaration to indicate that the method cannot be overridden by subclasses. The `Object` class does this—a number of its methods are `final`.

You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object. For example, you might want to make the `getFirstPlayer` method in this `ChessAlgorithm` class final:

```
class ChessAlgorithm {
    enum ChessPlayer { WHITE, BLACK }
    ...
    final ChessPlayer getFirstPlayer() {
        return ChessPlayer.WHITE;
    }
    ...
}
```

Methods called from constructors should generally be declared final. If a constructor calls a non-final method, a subclass may redefine that method with surprising or undesirable results.

Note that you can also declare an entire class final. A class that is declared final cannot be subclassed. This is particularly useful, for example, when creating an immutable class like the `String` class.

Abstract Methods and Classes

An *abstract class* is a class that is declared `abstract`—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

If a class includes abstract methods, then the class itself *must* be declared `abstract`, as in:

```
public abstract class GraphicObject {
   // declare fields
   // declare nonabstract methods
   abstract void draw();
}
```

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared `abstract`.

Note: Methods in an *interface* (see the Interfaces section) that are not declared as default or static are *implicitly* abstract, so the `abstract` modifier is not used with interface methods. (It can be used, but it is unnecessary.)

Abstract Classes Compared to Interfaces

Abstract classes are similar to interfaces. You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation. However, with abstract classes, you can declare fields that are not static and final, and define public, protected, and private concrete methods. With interfaces, all fields are automatically public, static, and final, and all methods that you declare or define (as default methods) are public. In addition, you can extend only one class, whether or not it is abstract, whereas you can implement any number of interfaces.

Which should you use, abstract classes or interfaces?

- Consider using abstract classes if any of these statements apply to your situation:
  - You want to share code among several closely related classes.
  - You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
  - You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.
- Consider using interfaces if any of these statements apply to your situation:
  - You expect that unrelated classes would implement your interface. For example, the

interfaces Comparable and Cloneable are implemented by many unrelated classes.
- You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
- You want to take advantage of multiple inheritance of type.

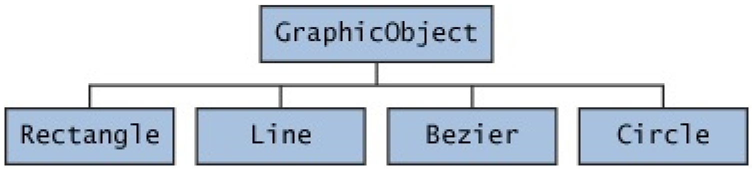
An example of an abstract class in the JDK is AbstractMap, which is part of the Collections Framework. Its subclasses (which include HashMap, TreeMap, and ConcurrentHashMap) share many methods (including get, put, isEmpty, containsKey, and containsValue) that AbstractMap defines.

An example of a class in the JDK that implements several interfaces is HashMap, which implements the interfaces Serializable, Cloneable, and Map<K, V>. By reading this list of interfaces, you can infer that an instance of HashMap (regardless of the developer or company who implemented the class) can be cloned, is serializable (which means that it can be converted into a byte stream; see the section Serializable Objects), and has the functionality of a map. In addition, the Map<K, V> interface has been enhanced with many default methods such as merge and forEach that older classes that have implemented this interface do not have to define.

Note that many software libraries use both abstract classes and interfaces; the HashMap class implements several interfaces and also extends the abstract class AbstractMap.

An Abstract Class Example

In an object-oriented drawing application, you can draw circles, rectangles, lines, Bezier curves, and many other graphic objects. These objects all have certain states (for example: position, orientation, line color, fill color) and behaviors (for example: moveTo, rotate, resize, draw) in common. Some of these states and behaviors are the same for all graphic objects (for example: position, fill color, and moveTo). Others require different implementations (for example, resize or draw). All GraphicObjects must be able to draw or resize themselves; they just differ in how they do it. This is a perfect situation for an abstract superclass. You can take advantage of the similarities and declare all the graphic objects to inherit from the same abstract parent object (for example, GraphicObject) as shown in the following figure.



Classes Rectangle, Line, Bezier, and Circle Inherit from GraphicObject

First, you declare an abstract class, GraphicObject, to provide member variables and methods that are wholly shared by all subclasses, such as the current position and the moveTo method. GraphicObject also declares abstract methods for methods, such as draw or resize, that need to be implemented by all subclasses but must be implemented in different ways. The GraphicObject class can look something like this:

```
abstract class GraphicObject {
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}
```

Each nonabstract subclass of `GraphicObject`, such as `Circle` and `Rectangle`, must provide implementations for the `draw` and `resize` methods:

```
class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

When an Abstract Class Implements an Interface

In the section on `Interfaces`, it was noted that a class that implements an interface must implement *all* of the interface's methods. It is possible, however, to define a class that does not implement all of the interface's methods, provided that the class is declared to be `abstract`. For example,

```
abstract class X implements Y {
  // implements all but one method of Y
}

class XX extends X {
  // implements the remaining method in Y
}
```

In this case, class `X` must be `abstract` because it does not fully implement `Y`, but class `XX` does, in fact, implement `Y`.

Class Members

An abstract class may have `static` fields and `static` methods. You can use these static members with a class reference (for example, `AbstractClass.staticMethod()`) as you would with any other class.

Summary of Inheritance

Except for the `Object` class, a class has exactly one direct superclass. A class inherits fields and methods from all its superclasses, whether direct or indirect. A subclass can override methods that it inherits, or it can hide fields or methods that it inherits. (Note that hiding fields is generally bad programming practice.)

The table in Overriding and Hiding Methods section shows the effect of declaring a method with the same signature as a method in the superclass.

The `Object` class is the top of the class hierarchy. All classes are descendants from this class and inherit methods from it. Useful methods inherited from `Object` include `toString()`, `equals()`, `clone()`, and `getClass()`.

You can prevent a class from being subclassed by using the `final` keyword in the class's declaration. Similarly, you can prevent a method from being overridden by subclasses by declaring it as a final method.

An abstract class can only be subclassed; it cannot be instantiated. An abstract class can contain abstract methods—methods that are declared but not implemented. Subclasses then provide the implementations for the abstract methods.

Questions and Exercises: Inheritance

Questions

1. Consider the following two classes:

```
public class ClassA {
    public void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public static void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}

public class ClassB extends ClassA {
    public static void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}
```

a. Which method overrides a method in the superclass?
b. Which method hides a method in the superclass?
c. What do the other methods do?

2. Consider the `Card`, `Deck`, and `DisplayDeck` classes you wrote in Questions and Exercises: Classes. What `Object` methods should each of these classes override?

Exercises

1. Write the implementations for the methods that you answered in question 2.

Check your answers.

Questions

Question 1: Consider the following two classes:

```
public class ClassA {
    public void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public static void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}

public class ClassB extends ClassA {
    public static void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}
```

Question 1a: Which method overrides a method in the superclass?
Answer 1a: `methodTwo`

Question 1b: Which method hides a method in the superclass?
Answer 1b: `methodFour`

Question 1c: What do the other methods do?
Answer 1c: They cause compile-time errors.

Question 2: Consider the `Card`, `Deck`, and `DisplayDeck` classes you wrote in the previous exercise. What `Object` methods should each of these classes override?
Answer 2: `Card` and `Deck` should override `equals`, `hashCode`, and `toString`.

Exercises

Exercise 1: Write the implementations for the methods that you answered in question 2.
Answer 1: See `Card2`.

Lesson: Numbers and Strings

Numbers

This section begins with a discussion of the `Number` class (in the `java.lang` package) and its subclasses. In particular, this section talks about the situations where you would use instantiations of these classes rather than the primitive data types. Additionally, this section talks about other classes you might need to work with numbers, such as formatting or using mathematical functions to complement the operators built into the language. Finally, there is a discussion on autoboxing and unboxing, a compiler feature that simplifies your code.

Strings

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects. This section describes using the `String` class to create and manipulate strings. It also compares the `String` and `StringBuilder` classes.

Note: See online version of topics in this ebook to download complete source code.

Numbers

This section begins with a discussion of the `Number` class in the `java.lang` package, its subclasses, and the situations where you would use instantiations of these classes rather than the primitive number types.

This section also presents the `PrintStream` and `DecimalFormat` classes, which provide methods for writing formatted numerical output.

Finally, the `Math` class in `java.lang` is discussed. It contains mathematical functions to complement the operators built into the language. This class has methods for the trigonometric functions, exponential functions, and so forth.

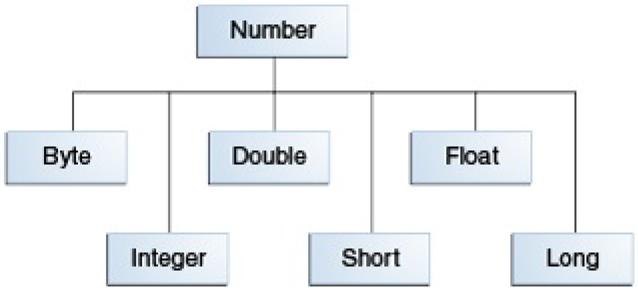
The Numbers Classes

When working with numbers, most of the time you use the primitive types in your code. For example:

```
int i = 500;
float gpa = 3.65f;
byte mask = 0xff;
```

There are, however, reasons to use objects in place of primitives, and the Java platform provides *wrapper* classes for each of the primitive data types. These classes "wrap" the primitive in an object. Often, the wrapping is done by the compiler—if you use a primitive where an object is expected, the compiler *boxes* the primitive in its wrapper class for you. Similarly, if you use a number object when a primitive is expected, the compiler *unboxes* the object for you. For more information, see Autoboxing and Unboxing

All of the numeric wrapper classes are subclasses of the abstract class `Number`:



Note: There are four other subclasses of `Number` that are not discussed here. `BigDecimal` and `BigInteger` are used for high-precision calculations. `AtomicInteger` and `AtomicLong` are used for multi-threaded applications.

There are three reasons that you might use a `Number` object rather than a primitive:

1. As an argument of a method that expects an object (often used when manipulating collections of numbers).
2. To use constants defined by the class, such as `MIN_VALUE` and `MAX_VALUE`, that provide the upper and lower bounds of the data type.
3. To use class methods for converting values to and from other primitive types, for converting to and from strings, and for converting between number systems (decimal, octal, hexadecimal, binary).

The following table lists the instance methods that all the subclasses of the `Number` class implement.

Methods Implemented by all Subclasses of Number

Method	Description
`byte byteValue()`	

`short shortValue()` `int intValue()` `long longValue()` `float floatValue()` `double doubleValue()`	Converts the value of this `Number` object to the primitive data type returned.
`int compareTo(Byte anotherByte)` `int compareTo(Double anotherDouble)` `int compareTo(Float anotherFloat)` `int compareTo(Integer anotherInteger)` `int compareTo(Long anotherLong)` `int compareTo(Short anotherShort)`	Compares this `Number` object to the argument.
`boolean equals(Object obj)`	Determines whether this number object is equal to the argument. The methods return `true` if the argument is not `null` and is an object of the same type and with the same numeric value. There are some extra requirements for `Double` and `Float` objects that are described in the Java API documentation.

Each `Number` class contains other methods that are useful for converting numbers to and from strings and for converting between number systems. The following table lists these methods in the `Integer` class. Methods for the other `Number` subclasses are similar:

Conversion Methods, Integer Class

Method	**Description**
`static Integer decode(String s)`	Decodes a string into an integer. Can accept string representations of decimal, octal, or hexadecimal numbers as input.
`static int parseInt(String s)`	Returns an integer (decimal only).
`static int parseInt(String s, int radix)`	Returns an integer, given a string representation of decimal, binary, octal, or hexadecimal (`radix` equals 10, 2, 8, or 16 respectively) numbers as input.
`String toString()`	Returns a `String` object representing the value of this `Integer`.
`static String toString(int i)`	Returns a `String` object representing the specified integer.
`static Integer valueOf(int i)`	Returns an `Integer` object holding the value of the specified primitive.
`static Integer valueOf(String s)`	Returns an `Integer` object holding the value of the specified string representation.

`static Integer valueOf(String s, int radix)`	Returns an `Integer` object holding the integer value of the specified string representation, parsed with the value of radix. For example, if s = "333" and radix = 8, the method returns the base-ten integer equivalent of the octal number 333.

Formatting Numeric Print Output

Earlier you saw the use of the `print` and `println` methods for printing strings to standard output (`System.out`). Since all numbers can be converted to strings (as you will see later in this lesson), you can use these methods to print out an arbitrary mixture of strings and numbers. The Java programming language has other methods, however, that allow you to exercise much more control over your print output when numbers are included.

The printf and format Methods

The `java.io` package includes a `PrintStream` class that has two formatting methods that you can use to replace `print` and `println`. These methods, `format` and `printf`, are equivalent to one another. The familiar `System.out` that you have been using happens to be a `PrintStream` object, so you can invoke `PrintStream` methods on `System.out`. Thus, you can use `format` or `printf` anywhere in your code where you have previously been using `print` or `println`. For example,

```
System.out.format(.....);
```

The syntax for these two `java.io.PrintStream` methods is the same:

```
public PrintStream format(String format, Object... args)
```

where `format` is a string that specifies the formatting to be used and `args` is a list of the variables to be printed using that formatting. A simple example would be

```
System.out.format("The value of " + "the float variable is " +
     "%f, while the value of the " + "integer variable is %d, " +
     "and the string is %s", floatVar, intVar, stringVar);
```

The first parameter, `format`, is a format string specifying how the objects in the second parameter, `args`, are to be formatted. The format string contains plain text as well as *format specifiers*, which are special characters that format the arguments of `Object... args`. (The notation `Object... args` is called *varargs*, which means that the number of arguments may vary.)

Format specifiers begin with a percent sign (%) and end with a *converter*. The converter is a character indicating the type of argument to be formatted. In between the percent sign (%) and the converter you can have optional flags and specifiers. There are many converters, flags, and specifiers, which are documented in `java.util.Formatter`

Here is a basic example:

```
int i = 461012;
System.out.format("The value of i is: %d%n", i);
```

The `%d` specifies that the single variable is a decimal integer. The `%n` is a platform-independent newline character. The output is:

```
The value of i is: 461012
```

The `printf` and `format` methods are overloaded. Each has a version with the following syntax:

```
public PrintStream format(Locale l, String format, Object... args)
```

To print numbers in the French system (where a comma is used in place of the decimal place in the English representation of floating point numbers), for example, you would use:

```
System.out.format(Locale.FRANCE,
    "The value of the float " + "variable is %f, while the " +
    "value of the integer variable " + "is %d, and the string is %s%n",
    floatVar, intVar, stringVar);
```

An Example

The following table lists some of the converters and flags that are used in the sample program, `TestFormat.java`, that follows the table.

Converters and Flags Used in TestFormat.java

Converter	Flag	Explanation
d		A decimal integer.
f		A float.
n		A new line character appropriate to the platform running the application. You should always use `%n`, rather than `\n`.
tB		A date & time conversion—locale-specific full name of month.
td, te		A date & time conversion—2-digit day of month. td has leading zeroes as needed, te does not.
ty, tY		A date & time conversion—ty = 2-digit year, tY = 4-digit year.
tl		A date & time conversion—hour in 12-hour clock.
tM		A date & time conversion—minutes in 2 digits, with leading zeroes as necessary.

tp		A date & time conversion—locale-specific am/pm (lower case).
tm		A date & time conversion—months in 2 digits, with leading zeroes as necessary.
tD		A date & time conversion—date as %tm%td%ty
	08	Eight characters in width, with leading zeroes as necessary.
	+	Includes sign, whether positive or negative.
	,	Includes locale-specific grouping characters.
	-	Left-justified..
	.3	Three places after decimal point.
	10.3	Ten characters in width, right justified, with three places after decimal point.

The following program shows some of the formatting that you can do with `format`. The output is shown within double quotes in the embedded comment:

```
import java.util.Calendar;
import java.util.Locale;

public class TestFormat {

    public static void main(String[] args) {
      long n = 461012;
      System.out.format("%d%n", n);      //  -->  "461012"
      System.out.format("%08d%n", n);    //  -->  "00461012"
      System.out.format("%+8d%n", n);    //  -->  " +461012"
      System.out.format("%,8d%n", n);    // -->  " 461,012"
      System.out.format("%+,8d%n%n", n); //  -->  "+461,012"

      double pi = Math.PI;

      System.out.format("%f%n", pi);       // -->  "3.141593"
      System.out.format("%.3f%n", pi);     // -->  "3.142"
      System.out.format("%10.3f%n", pi);   // -->  "     3.142"
      System.out.format("%-10.3f%n", pi);  // -->  "3.142"
      System.out.format(Locale.FRANCE,
                        "%-10.4f%n%n", pi); // -->  "3,1416"

      Calendar c = Calendar.getInstance();
      System.out.format("%tB %te, %tY%n", c, c, c); // -->  "May 29, 2006"

      System.out.format("%tl:%tM %tp%n", c, c, c);  // -->  "2:34 am"

      System.out.format("%tD%n", c);    // -->  "05/29/06"
    }
}
```

Note: The discussion in this section covers just the basics of the `format` and `printf` methods. Further detail can be found in the Basic I/O section of the Essential trail, in the "Formatting" page. Using `String.format` to create strings is covered in Strings.

The DecimalFormat Class

You can use the [java.text.DecimalFormat]class to control the display of leading and trailing zeros, prefixes and suffixes, grouping (thousands) separators, and the decimal separator. `DecimalFormat` offers a great deal of flexibility in the formatting of numbers, but it can make your code more complex.

The example that follows creates a `DecimalFormat` object, `myFormatter`, by passing a pattern string to the `DecimalFormat` constructor. The `format()` method, which `DecimalFormat` inherits from `NumberFormat`, is then invoked by `myFormatter`—it accepts a `double` value as an argument and returns the formatted number in a string:

Here is a sample program that illustrates the use of `DecimalFormat`:

```
import java.text.*;

public class DecimalFormatDemo {

   static public void customFormat(String pattern, double value ) {
      DecimalFormat myFormatter = new DecimalFormat(pattern);
      String output = myFormatter.format(value);
      System.out.println(value + "  " + pattern + "  " + output);
   }

   static public void main(String[] args) {

      customFormat("###,###.###", 123456.789);
      customFormat("###.##", 123456.789);
      customFormat("000000.000", 123.78);
      customFormat("$###,###.###", 12345.67);
   }
}
```

The output is:

```
123456.789  ###,###.###  123,456.789
123456.789  ###.##  123456.79
123.78  000000.000  000123.780
12345.67  $###,###.###  $12,345.67
```

The following table explains each line of output.

DecimalFormat.java Output

Value	Pattern	Output	Explanation
123456.789	###,###.###	123,456.789	The pound sign (#) denotes a digit, the comma is a placeholder for the grouping separator, and the period is a

			placeholder for the decimal separator.
123456.789	###.##	123456.79	The `value` has three digits to the right of the decimal point, but the `pattern` has only two. The `format` method handles this by rounding up.
123.78	000000.000	000123.780	The `pattern` specifies leading and trailing zeros, because the 0 character is used instead of the pound sign (#).
12345.67	$###,###.###	$12,345.67	The first character in the `pattern` is the dollar sign ($). Note that it immediately precedes the leftmost digit in the formatted `output`.

Beyond Basic Arithmetic

The Java programming language supports basic arithmetic with its arithmetic operators: +, -, *, /, and %. The `Math` class in the `java.lang` package provides methods and constants for doing more advanced mathematical computation.

The methods in the `Math` class are all static, so you call them directly from the class, like this:

```
Math.cos(angle);
```

Note: Using the `static import` language feature, you don't have to write `Math` in front of every math function:

```
import static java.lang.Math.*;
```

This allows you to invoke the `Math` class methods by their simple names. For example:

```
cos(angle);
```

Constants and Basic Methods

The `Math` class includes two constants:

- `Math.E`, which is the base of natural logarithms, and
- `Math.PI`, which is the ratio of the circumference of a circle to its diameter.

The `Math` class also includes more than 40 static methods. The following table lists a number of the basic methods.

Basic Math Methods

`double abs(double d)`
`float abs(float f)`
`int abs(int i)`
`long abs(long lng)`
	Returns the absolute value of the argument.

`double ceil(double d)`	Returns the smallest integer that is greater than or equal to the argument. Returned as a double.
`double floor(double d)`	Returns the largest integer that is less than or equal to the argument. Returned as a double.
`double rint(double d)`	Returns the integer that is closest in value to the argument. Returned as a double.
`long round(double d)`	Returns the closest long or int, as indicated by the method's return

`int round(float f)`	type, to the argument.

`double min(double arg1, double arg2)`
`float min(float arg1, float arg2)`
`int min(int arg1, int arg2)`
`long min(long arg1, long arg2)`
	Returns the smaller of the two arguments.

`double max(double arg1, double arg2)`
`float max(float arg1, float arg2)`
`int max(int arg1, int arg2)`
`long max(long arg1, long arg2)`
	Returns the larger of the two arguments.

The following program, `BasicMathDemo` , illustrates how to use some of these methods:

```
public class BasicMathDemo {
    public static void main(String[] args) {
        double a = -191.635;
        double b = 43.74;
        int c = 16, d = 45;

        System.out.printf("The absolute value " + "of %.3f is %.3f%n",
                          a, Math.abs(a));

        System.out.printf("The ceiling of " + "%.2f is %.0f%n",
                          b, Math.ceil(b));

        System.out.printf("The floor of " + "%.2f is %.0f%n",
                          b, Math.floor(b));

        System.out.printf("The rint of %.2f " + "is %.0f%n",
                          b, Math.rint(b));

        System.out.printf("The max of %d and " + "%d is %d%n",
                          c, d, Math.max(c, d));

        System.out.printf("The min of of %d " + "and %d is %d%n",
                          c, d, Math.min(c, d));
    }
}
```

Here's the output from this program:

```
The absolute value of -191.635 is 191.635
The ceiling of 43.74 is 44
The floor of 43.74 is 43
The rint of 43.74 is 44
The max of 16 and 45 is 45
The min of 16 and 45 is 16
```

Exponential and Logarithmic Methods

The next table lists exponential and logarithmic methods of the `Math` class.

Exponential and Logarithmic Methods

Method	Description
`double exp(double d)`	Returns the base of the natural logarithms, e, to the power of the argument.
`double log(double d)`	Returns the natural logarithm of the argument.
`double pow(double base, double exponent)`	Returns the value of the first argument raised to the power of the second argument.
`double sqrt(double d)`	Returns the square root of the argument.

The following program, `ExponentialDemo`, displays the value of `e`, then calls each of the methods listed in the previous table on arbitrarily chosen numbers:

```
public class ExponentialDemo {
    public static void main(String[] args) {
        double x = 11.635;
        double y = 2.76;

        System.out.printf("The value of " + "e is %.4f%n",
                          Math.E);

        System.out.printf("exp(%.3f) " + "is %.3f%n",
                          x, Math.exp(x));

        System.out.printf("log(%.3f) is " + "%.3f%n",
                          x, Math.log(x));

        System.out.printf("pow(%.3f, %.3f) " + "is %.3f%n",
                          x, y, Math.pow(x, y));

        System.out.printf("sqrt(%.3f) is " + "%.3f%n",
                          x, Math.sqrt(x));
    }
}
```

Here's the output you'll see when you run `ExponentialDemo`:

```
The value of e is 2.7183
exp(11.635) is 112983.831
log(11.635) is 2.454
pow(11.635, 2.760) is 874.008
sqrt(11.635) is 3.411
```

Trigonometric Methods

The `Math` class also provides a collection of trigonometric functions, which are summarized in the following table. The value passed into each of these methods is an angle expressed in radians. You can use the `toRadians` method to convert from degrees to radians.

Trigonometric Methods

Method	Description
`double sin(double d)`	Returns the sine of the specified double value.
`double cos(double d)`	Returns the cosine of the specified double value.
`double tan(double d)`	Returns the tangent of the specified double value.
`double asin(double d)`	Returns the arcsine of the specified double value.
`double acos(double d)`	Returns the arccosine of the specified double value.
`double atan(double d)`	Returns the arctangent of the specified double value.
`double atan2(double y, double x)`	Converts rectangular coordinates `(x, y)` to polar coordinate `(r, theta)` and returns `theta`.

`double toDegrees(double d)`
`double toRadians(double d)`
	Converts the argument to degrees or radians.

Here's a program, `TrigonometricDemo`, that uses each of these methods to compute various trigonometric values for a 45-degree angle:

```
public class TrigonometricDemo {
    public static void main(String[] args) {
        double degrees = 45.0;
        double radians = Math.toRadians(degrees);

        System.out.format("The value of pi " + "is %.4f%n",
                          Math.PI);

        System.out.format("The sine of %.1f " + "degrees is %.4f%n",
                         degrees, Math.sin(radians));

        System.out.format("The cosine of %.1f " + "degrees is %.4f%n",
                         degrees, Math.cos(radians));

        System.out.format("The tangent of %.1f " + "degrees is %.4f%n",
                         degrees, Math.tan(radians));

        System.out.format("The arcsine of %.4f " + "is %.4f degrees %n",
                         Math.sin(radians),
                         Math.toDegrees(Math.asin(Math.sin(radians))));

        System.out.format("The arccosine of %.4f " + "is %.4f degrees %n",
                         Math.cos(radians),
                         Math.toDegrees(Math.acos(Math.cos(radians))));

        System.out.format("The arctangent of %.4f " + "is %.4f degrees %n",
                         Math.tan(radians),
                         Math.toDegrees(Math.atan(Math.tan(radians))));
    }
}
```

The output of this program is as follows:

```
The value of pi is 3.1416
The sine of 45.0 degrees is 0.7071
The cosine of 45.0 degrees is 0.7071
The tangent of 45.0 degrees is 1.0000
The arcsine of 0.7071 is 45.0000 degrees
The arccosine of 0.7071 is 45.0000 degrees
The arctangent of 1.0000 is 45.0000 degrees
```

Random Numbers

The `random()` method returns a pseudo-randomly selected number between 0.0 and 1.0. The range includes 0.0 but not 1.0. In other words: `0.0 <= Math.random() < 1.0`. To get a number in a different range, you can perform arithmetic on the value returned by the random method. For example, to generate an integer between 0 and 9, you would write:

```
int number = (int)(Math.random() * 10);
```

By multiplying the value by 10, the range of possible values becomes `0.0 <= number < 10.0`.

Using `Math.random` works well when you need to generate a single random number. If you need to generate a series of random numbers, you should create an instance of `java.util.Random` and invoke methods on that object to generate numbers.

Summary of Numbers

You use one of the wrapper classes `Byte`, `Double`, `Float`, `Integer`, `Long`, or `Short` to wrap a number of primitive type in an object. The Java compiler automatically wraps (boxes) primitives for you when necessary and unboxes them, again when necessary.

The `Number` classes include constants and useful class methods. The `MIN_VALUE` and `MAX_VALUE` constants contain the smallest and largest values that can be contained by an object of that type. The `byteValue`, `shortValue`, and similar methods convert one numeric type to another. The `valueOf` method converts a string to a number, and the `toString` method converts a number to a string.

To format a string containing numbers for output, you can use the `printf()` or `format()` methods in the `PrintStream` class. Alternatively, you can use the `NumberFormat` class to customize numerical formats using patterns.

The `Math` class contains a variety of class methods for performing mathematical functions, including exponential, logarithmic, and trigonometric methods. `Math` also includes basic arithmetic functions, such as absolute value and rounding, and a method, `random()`, for generating random numbers.

Questions and Exercises: Numbers

Questions

1. Use the API documentation to find the answers to the following questions:
a. What `Integer` method can you use to convert an `int` into a string that expresses the number in hexadecimal? For example, what method converts the integer 65 into the string "41"?
b. What `Integer` method would you use to convert a string expressed in base 5 into the equivalent `int`? For example, how would you convert the string "230" into the integer value 65? Show the code you would use to accomplish this task.
c. What Double method can you use to detect whether a floating-point number has the special value Not a Number (`NaN`)?
2. What is the value of the following expression, and why?

```
Integer.valueOf(1).equals(Long.valueOf(1))
```

Exercises

1. Change `MaxVariablesDemo` to show minimum values instead of maximum values. You can delete all code related to the variables `aChar` and `aBoolean`. What is the output?
2. Create a program that reads an unspecified number of integer arguments from the command line and adds them together. For example, suppose that you enter the following:

```
java Adder 1 3 2 10
```

The program should display `16` and then exit. The program should display an error message if the user enters only one argument. You can base your program on `ValueOfDemo`.
3. Create a program that is similar to the previous one but has the following differences:

- Instead of reading integer arguments, it reads floating-point arguments.
- It displays the sum of the arguments, using exactly two digits to the right of the decimal point.

For example, suppose that you enter the following:

```
java FPAdder 1 1e2 3.0 4.754
```

The program would display `108.75`. Depending on your locale, the decimal point might be a comma (`,`) instead of a period (`.`).

Check your answers.

Questions

1. Use the API documentation to find the answers to the following questions:
a. Question: What `Integer` method can you use to convert an `int` into a string that expresses the number in hexadecimal? For example, what method converts the integer 65 into the string "41"?
Answer: `toHexString`
b. Question: What `Integer` method would you use to convert a string expressed in base 5 into the equivalent `int`? For example, how would you convert the string "230" into the integer value 65? Show the code you would use to accomplish this task.
Answer: `valueOf`. Here's how:

```
String base5String = "230";
int result = Integer.valueOf(base5String, 5);
```

c. Question: What Double method can you use to detect whether a floating-point number has the special value Not a Number (`NaN`)?
Answer: `isNaN`
2. Question: What is the value of the following expression, and why?

```
Integer.valueOf(1).equals(Long.valueOf(1))
```

Answer: False. The two objects (the `Integer` and the `Long`) have different types.

Exercises

1. Exercise: Change `MaxVariablesDemo` to show minimum values instead of maximum values. You can delete all code related to the variables `aChar` and `aBoolean`. What is the output?
Answer: See `MinVariablesDemo`. Here is the output:

```
The smallest byte value is -128
The smallest short value is -32768
The smallest integer value is -2147483648
The smallest long value is -9223372036854775808
The smallest float value is 1.4E-45
The smallest double value is 4.9E-324
```

2. Exercise: Create a program that reads an unspecified number of integer arguments from the command line and adds them together. For example, suppose that you enter the following:

```
       java Adder 1 3 2 10
```

The program should display `16` and then exit. The program should display an error message if the user enters only one argument. You can base your program on `ValueOfDemo`.
Answer: See `Adder`.
3. Exercise: Create a program that is similar to the previous one but has the following differences:

- Instead of reading integer arguments, it reads floating-point arguments.
- It displays the sum of the arguments, using exactly two digits to the right of the decimal point.

For example, suppose that you enter the following:

```
java FPAdder 1 1e2 3.0 4.754
```

The program would display `108.75`. Depending on your locale, the decimal point might be a comma (`,`) instead of a period (`.`).

Answer: See `FPAdder`.

Characters

Most of the time, if you are using a single character value, you will use the primitive `char` type. For example:

```
char ch = 'a';
// Unicode for uppercase Greek omega character
char uniChar = '\u03A9';
// an array of chars
char[] charArray = { 'a', 'b', 'c', 'd', 'e' };
```

There are times, however, when you need to use a char as an object—for example, as a method argument where an object is expected. The Java programming language provides a *wrapper* class that "wraps" the `char` in a `Character` object for this purpose. An object of type `Character` contains a single field, whose type is `char`. This Character class also offers a number of useful class (i.e., static) methods for manipulating characters.

You can create a `Character` object with the `Character` constructor:

```
Character ch = new Character('a');
```

The Java compiler will also create a `Character` object for you under some circumstances. For example, if you pass a primitive `char` into a method that expects an object, the compiler automatically converts the `char` to a `Character` for you. This feature is called *autoboxing*—or *unboxing*, if the conversion goes the other way. For more information on autoboxing and unboxing, see Autoboxing and Unboxing.

Note: The `Character` class is immutable, so that once it is created, a `Character` object cannot be changed.

The following table lists some of the most useful methods in the `Character` class, but is not exhaustive. For a complete listing of all methods in this class (there are more than 50), refer to the java.lang.Character API specification.

Useful Methods in the Character Class

Method	Description

`boolean isLetter(char ch)`
`boolean isDigit(char ch)`
	Determines whether the specified char value is a letter or a digit, respectively.

`boolean isWhitespace(char ch)`	Determines whether the specified char value is white space.

`boolean isUpperCase(char ch)`
`boolean isLowerCase(char ch)`
	Determines whether the specified char value is uppercase or lowercase, respectively.

`char toUpperCase(char ch)`
`char toLowerCase(char ch)`
	Returns the uppercase or lowercase form of the specified char value.

`toString(char ch)`	Returns a `String` object representing the specified character value — that is, a one-character string.

Escape Sequences

A character preceded by a backslash (\) is an *escape sequence* and has special meaning to the compiler. The following table shows the Java escape sequences:

Escape Sequences

Escape Sequence	**Description**
`\t`	Insert a tab in the text at this point.
`\b`	Insert a backspace in the text at this point.
`\n`	Insert a newline in the text at this point.
`\r`	Insert a carriage return in the text at this point.
`\f`	Insert a formfeed in the text at this point.
`\'`	Insert a single quote character in the text at this point.
`\"`	Insert a double quote character in the text at this point.
`\\`	Insert a backslash character in the text at this point.

When an escape sequence is encountered in a print statement, the compiler interprets it accordingly. For example, if you want to put quotes within quotes you must use the escape sequence, \", on the interior quotes. To print the sentence

```
She said "Hello!" to me.
```

you would write

```
System.out.println("She said \"Hello!\" to me.");
```

Strings

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects.

The Java platform provides the `String` class to create and manipulate strings.

Creating Strings

The most direct way to create a string is to write:

```
String greeting = "Hello world!";
```

In this case, "Hello world!" is a *string literal*—a series of characters in your code that is enclosed in double quotes. Whenever it encounters a string literal in your code, the compiler creates a `String` object with its value—in this case, `Hello world!`.

As with any other object, you can create `String` objects by using the `new` keyword and a constructor. The `String` class has thirteen constructors that allow you to provide the initial value of the string using different sources, such as an array of characters:

```
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
String helloString = new String(helloArray);
System.out.println(helloString);
```

The last line of this code snippet displays `hello`.

Note: The `String` class is immutable, so that once it is created a `String` object cannot be changed. The `String` class has a number of methods, some of which will be discussed below, that appear to modify strings. Since strings are immutable, what these methods really do is create and return a new string that contains the result of the operation.

String Length

Methods used to obtain information about an object are known as *accessor methods*. One accessor method that you can use with strings is the `length()` method, which returns the number of characters contained in the string object. After the following two lines of code have been executed, `len` equals 17:

```
String palindrome = "Dot saw I was Tod";
int len = palindrome.length();
```

A *palindrome* is a word or sentence that is symmetric—it is spelled the same forward and backward, ignoring case and punctuation. Here is a short and inefficient program to reverse a palindrome string.

It invokes the `String` method `charAt(i)`, which returns the ith character in the string, counting from 0.

```
public class StringDemo {
    public static void main(String[] args) {
        String palindrome = "Dot saw I was Tod";
        int len = palindrome.length();
        char[] tempCharArray = new char[len];
        char[] charArray = new char[len];

        // put original string in an
        // array of chars
        for (int i = 0; i < len; i++) {
            tempCharArray[i] =
                palindrome.charAt(i);
        }

        // reverse array of chars
        for (int j = 0; j < len; j++) {
            charArray[j] =
                tempCharArray[len - 1 - j];
        }

        String reversePalindrome =
            new String(charArray);
        System.out.println(reversePalindrome);
    }
}
```

Running the program produces this output:

```
doT saw I was toD
```

To accomplish the string reversal, the program had to convert the string to an array of characters (first `for` loop), reverse the array into a second array (second `for` loop), and then convert back to a string. The `String` class includes a method, `getChars()`, to convert a string, or a portion of a string, into an array of characters so we could replace the first `for` loop in the program above with

```
palindrome.getChars(0, len, tempCharArray, 0);
```

Concatenating Strings

The `String` class includes a method for concatenating two strings:

```
string1.concat(string2);
```

This returns a new string that is string1 with string2 added to it at the end.

You can also use the `concat()` method with string literals, as in:

```
"My name is ".concat("Rumplestiltskin");
```

Strings are more commonly concatenated with the **+** operator, as in

```
"Hello," + " world" + "!"
```

which results in

```
"Hello, world!"
```

The **+** operator is widely used in `print` statements. For example:

```
String string1 = "saw I was ";
System.out.println("Dot " + string1 + "Tod");
```

which prints

```
Dot saw I was Tod
```

Such a concatenation can be a mixture of any objects. For each object that is not a `String`, its `toString()` method is called to convert it to a `String`.

Note: The Java programming language does not permit literal strings to span lines in source files, so you must use the + concatenation operator at the end of each line in a multi-line string. For example:

```
String quote =
    "Now is the time for all good " +
    "men to come to the aid of their country.";
```

Breaking strings between lines using the + concatenation operator is, once again, very common in `print` statements.

Creating Format Strings

You have seen the use of the `printf()` and `format()` methods to print output with formatted numbers. The `String` class has an equivalent class method, `format()`, that returns a `String` object rather than a `PrintStream` object.

Using `String's` static `format()` method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement. For example, instead of

```
System.out.printf("The value of the float " +
                  "variable is %f, while " +
                  "the value of the " +
                  "integer variable is %d, " +
                  "and the string is %s",
                  floatVar, intVar, stringVar);
```

you can write

```
String fs;
fs = String.format("The value of the float " +
                   "variable is %f, while " +
                   "the value of the " +
                   "integer variable is %d, " +
                   " and the string is %s",
                   floatVar, intVar, stringVar);
System.out.println(fs);
```

Converting Between Numbers and Strings

Converting Strings to Numbers

Frequently, a program ends up with numeric data in a string object—a value entered by the user, for example.

The `Number` subclasses that wrap primitive numeric types (`Byte`, `Integer`, `Double`, `Float`, `Long`, and `Short`) each provide a class method named `valueOf` that converts a string to an object of that type. Here is an example, `ValueOfDemo` , that gets two strings from the command line, converts them to numbers, and performs arithmetic operations on the values:

```
public class ValueOfDemo {
    public static void main(String[] args) {

        // this program requires two
        // arguments on the command line
        if (args.length == 2) {
            // convert strings to numbers
            float a = (Float.valueOf(args[0])).floatValue();
            float b = (Float.valueOf(args[1])).floatValue();

            // do some arithmetic
            System.out.println("a + b = " +
                               (a + b));
            System.out.println("a - b = " +
                               (a - b));
            System.out.println("a * b = " +
                               (a * b));
            System.out.println("a / b = " +
                               (a / b));
            System.out.println("a % b = " +
                               (a % b));
        } else {
            System.out.println("This program " +
                "requires two command-line arguments.");
        }
    }
}
```

The following is the output from the program when you use `4.5` and `87.2` for the command-line arguments:

```
a + b = 91.7
a - b = -82.7
a * b = 392.4
a / b = 0.0516055
a % b = 4.5
```

Note: Each of the `Number` subclasses that wrap primitive numeric types also provides a `parseXXXX()` method (for example, `parseFloat()`) that can be used to convert strings to primitive numbers. Since a primitive type is returned instead of an object, the `parseFloat()` method is more direct than the `valueOf()` method. For example, in the `ValueOfDemo` program, we could use:

```
float a = Float.parseFloat(args[0]);
float b = Float.parseFloat(args[1]);
```

Converting Numbers to Strings

Sometimes you need to convert a number to a string because you need to operate on the value in its string form. There are several easy ways to convert a number to a string:

```
int i;
// Concatenate "i" with an empty string; conversion is handled for you.
String s1 = "" + i;
```

or

```
// The valueOf class method.
String s2 = String.valueOf(i);
```

Each of the `Number` subclasses includes a class method, `toString()`, that will convert its primitive type to a string. For example:

```
int i;
double d;
String s3 = Integer.toString(i);
String s4 = Double.toString(d);
```

The `ToStringDemo` example uses the `toString` method to convert a number to a string. The program then uses some string methods to compute the number of digits before and after the decimal point:

```
public class ToStringDemo {

    public static void main(String[] args) {
        double d = 858.48;
        String s = Double.toString(d);

        int dot = s.indexOf('.');

        System.out.println(dot + " digits " +
            "before decimal point.");
        System.out.println( (s.length() - dot - 1) +
            " digits after decimal point.");
    }
}
```

The output of this program is:

```
3 digits before decimal point.
2 digits after decimal point.
```

Manipulating Characters in a String

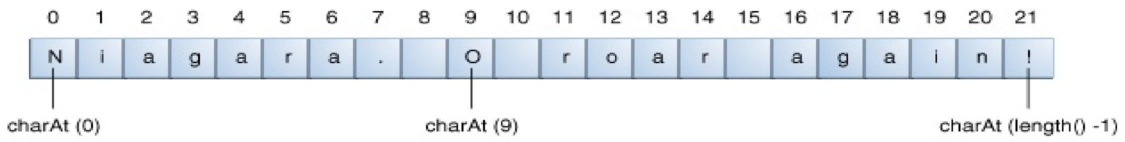
The `String` class has a number of methods for examining the contents of strings, finding characters or substrings within a string, changing case, and other tasks.

Getting Characters and Substrings by Index

You can get the character at a particular index within a string by invoking the `charAt()` accessor method. The index of the first character is 0, while the index of the last character is `length()-1`. For example, the following code gets the character at index 9 in a string:

```
String anotherPalindrome = "Niagara. O roar again!";
char aChar = anotherPalindrome.charAt(9);
```

Indices begin at 0, so the character at index 9 is 'O', as illustrated in the following figure:



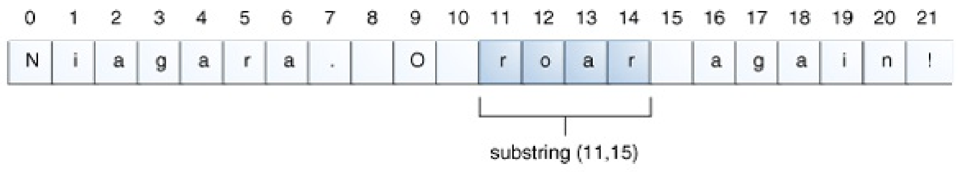
If you want to get more than one consecutive character from a string, you can use the `substring` method. The `substring` method has two versions, as shown in the following table:

The substring Methods in the String Class

Method	Description
`String substring(int beginIndex, int endIndex)`	Returns a new string that is a substring of this string. The first integer argument specifies the index of the first character. The second integer argument is the index of the last character - 1.
`String substring(int beginIndex)`	Returns a new string that is a substring of this string. The integer argument specifies the index of the first character. Here, the returned substring extends to the end of the original string.

The following code gets from the Niagara palindrome the substring that extends from index 11 up to, but not including, index 15, which is the word "roar":

```
String anotherPalindrome = "Niagara. O roar again!";
String roar = anotherPalindrome.substring(11, 15);
```



Other Methods for Manipulating Strings

Here are several other `String` methods for manipulating strings:

Other Methods in the String Class for Manipulating Strings

Method	Description

`String[] split(String regex)`
`String[] split(String regex, int limit)`
	Searches for a match as specified by the string argument (which contains a regular expression) and splits this string into an array of strings accordingly. The optional integer argument specifies the maximum size of the returned array. Regular expressions are covered in the lesson titled "Regular Expressions."

`CharSequence subSequence(int beginIndex, int endIndex)`	Returns a new character sequence constructed from `beginIndex` index up until `endIndex` - 1.
`String trim()`	Returns a copy of this string with leading and trailing white space removed.

`String toLowerCase()`
`String toUpperCase()`
	Returns a copy of this string converted to lowercase or uppercase. If no conversions are necessary, these methods return the original string.

Searching for Characters and Substrings in a String

Here are some other `String` methods for finding characters or substrings within a string. The `String` class provides accessor methods that return the position within the string of a specific character or substring: `indexOf()` and `lastIndexOf()`. The `indexOf()` methods search forward from the beginning of the string, and the `lastIndexOf()` methods search backward from the end of the string. If a character or substring is not found, `indexOf()` and `lastIndexOf()` return -1.

The `String` class also provides a search method, `contains`, that returns true if the string contains a particular character sequence. Use this method when you only need to know that the string contains a character sequence, but the precise location isn't important.

The following table describes the various string search methods.

The Search Methods in the String Class

Method	Description

`int indexOf(int ch)`
`int lastIndexOf(int ch)`
	Returns the index of the first (last) occurrence of the specified character.

`int indexOf(int ch, int fromIndex)`
`int lastIndexOf(int ch, int fromIndex)`
	Returns the index of the first (last) occurrence of the specified character, searching forward (backward) from the specified index.

`int indexOf(String str)`
`int lastIndexOf(String str)`
	Returns the index of the first (last) occurrence of the specified substring.

`int indexOf(String str, int fromIndex)`
`int lastIndexOf(String str, int fromIndex)`
	Returns the index of the first (last) occurrence of the specified substring, searching forward (backward) from the specified index.

`boolean contains(CharSequence s)`	Returns true if the string contains the specified character sequence.

Note:`CharSequence` is an interface that is implemented by the `String` class. Therefore, you can use a string as an argument for the `contains()` method.

Replacing Characters and Substrings into a String

The `String` class has very few methods for inserting characters or substrings into a string. In general, they are not needed: You can create a new string by concatenation of substrings you have *removed* from a string with the substring that you want to insert.

The `String` class does have four methods for *replacing* found characters or substrings, however. They are:

Methods in the String Class for Manipulating Strings

Method	Description
`String replace(char oldChar, char newChar)`	Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
`String replace(CharSequence target, CharSequence replacement)`	Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.
`String replaceAll(String regex, String replacement)`	Replaces each substring of this string that matches the given regular expression with the given replacement.

`String replaceFirst(String regex, String replacement)`	Replaces the first substring of this string that matches the given regular expression with the given replacement.

An Example

The following class, `Filename`, illustrates the use of `lastIndexOf()` and `substring()` to isolate different parts of a file name.

Note: The methods in the following `Filename` class don't do any error checking and assume that their argument contains a full directory path and a filename with an extension. If these methods were production code, they would verify that their arguments were properly constructed.

```
public class Filename {
    private String fullPath;
    private char pathSeparator,
                 extensionSeparator;

    public Filename(String str, char sep, char ext) {
        fullPath = str;
        pathSeparator = sep;
        extensionSeparator = ext;
    }

    public String extension() {
        int dot = fullPath.lastIndexOf(extensionSeparator);
        return fullPath.substring(dot + 1);
    }

    // gets filename without extension
    public String filename() {
        int dot = fullPath.lastIndexOf(extensionSeparator);
        int sep = fullPath.lastIndexOf(pathSeparator);
        return fullPath.substring(sep + 1, dot);
    }

    public String path() {
        int sep = fullPath.lastIndexOf(pathSeparator);
        return fullPath.substring(0, sep);
    }
}
```

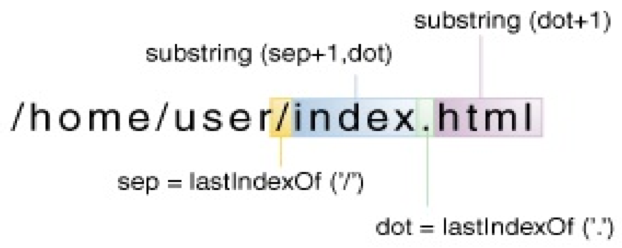
Here is a program, `FilenameDemo`, that constructs a `Filename` object and calls all of its methods:

```
public class FilenameDemo {
    public static void main(String[] args) {
        final String FPATH = "/home/user/index.html";
        Filename myHomePage = new Filename(FPATH, '/', '.');
        System.out.println("Extension = " + myHomePage.extension());
        System.out.println("Filename = " + myHomePage.filename());
        System.out.println("Path = " + myHomePage.path());
    }
}
```

And here's the output from the program:

```
Extension = html
Filename = index
Path = /home/user
```

As shown in the following figure, our `extension` method uses `lastIndexOf` to locate the last occurrence of the period (.) in the file name. Then `substring` uses the return value of `lastIndexOf` to extract the file name extension — that is, the substring from the period to the end of the string. This code assumes that the file name has a period in it; if the file name does not have a period, `lastIndexOf` returns -1, and the substring method throws a `StringIndexOutOfBoundsException`.



Also, notice that the `extension` method uses `dot + 1` as the argument to `substring`. If the period character (.) is the last character of the string, `dot + 1` is equal to the length of the string, which is one larger than the largest index into the string (because indices start at 0). This is a legal argument to `substring` because that method accepts an index equal to, but not greater than, the length of the string and interprets it to mean "the end of the string."

Comparing Strings and Portions of Strings

The `String` class has a number of methods for comparing strings and portions of strings. The following table lists these methods.

Methods for Comparing Strings

Method	Description
`boolean endsWith(String suffix)` `boolean startsWith(String prefix)`	Returns `true` if this string ends with or begins with the substring specified as an argument to the method.
`boolean startsWith(String prefix, int offset)`	Considers the string beginning at the index `offset`, and returns `true` if it begins with the substring specified as an argument.
`int compareTo(String anotherString)`	Compares two strings lexicographically. Returns an integer indicating whether this string is greater than (result is > 0), equal to (result is = 0), or less than (result is < 0) the argument.
`int compareToIgnoreCase(String str)`	Compares two strings lexicographically, ignoring differences in case. Returns an integer indicating whether this string is greater than (result is > 0), equal to (result is = 0), or less than (result is < 0) the argument.
`boolean equals(Object anObject)`	Returns `true` if and only if the argument is a `String` object that represents the same sequence of characters as this object.
`boolean equalsIgnoreCase(String anotherString)`	Returns `true` if and only if the argument is a `String` object that represents the same sequence of characters as this object, ignoring differences in case.
`boolean regionMatches(int toffset, String other, int ooffset, int len)`	Tests whether the specified region of this string matches the specified region of the String argument. Region is of length `len` and begins at the index `toffset` for this string and `ooffset` for the other string.
	Tests whether the specified region of this string matches the specified region of the String argument.

`boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)`	Region is of length `len` and begins at the index `toffset` for this string and `ooffset` for the other string. The boolean argument indicates whether case should be ignored; if true, case is ignored when comparing characters.
`boolean matches(String regex)`	Tests whether this string matches the specified regular expression. Regular expressions are discussed in the lesson titled "Regular Expressions."

The following program, `RegionMatchesDemo`, uses the `regionMatches` method to search for a string within another string:

```
public class RegionMatchesDemo {
    public static void main(String[] args) {
        String searchMe = "Green Eggs and Ham";
        String findMe = "Eggs";
        int searchMeLength = searchMe.length();
        int findMeLength = findMe.length();
        boolean foundIt = false;
        for (int i = 0;
             i <= (searchMeLength - findMeLength);
             i++) {
           if (searchMe.regionMatches(i, findMe, 0, findMeLength)) {
              foundIt = true;
              System.out.println(searchMe.substring(i, i + findMeLength));
              break;
           }
        }
        if (!foundIt)
            System.out.println("No match found.");
    }
}
```

The output from this program is `Eggs`.

The program steps through the string referred to by `searchMe` one character at a time. For each character, the program calls the regionMatches method to determine whether the substring beginning with the current character matches the string the program is looking for.

The StringBuilder Class

`StringBuilder` objects are like `String` objects, except that they can be modified. Internally, these objects are treated like variable-length arrays that contain a sequence of characters. At any point, the length and content of the sequence can be changed through method invocations.

Strings should always be used unless string builders offer an advantage in terms of simpler code (see the sample program at the end of this section) or better performance. For example, if you need to concatenate a large number of strings, appending to a `StringBuilder` object is more efficient.

Length and Capacity

The `StringBuilder` class, like the `String` class, has a `length()` method that returns the length of the character sequence in the builder.

Unlike strings, every string builder also has a *capacity*, the number of character spaces that have been allocated. The capacity, which is returned by the `capacity()` method, is always greater than or equal to the length (usually greater than) and will automatically expand as necessary to accommodate additions to the string builder.

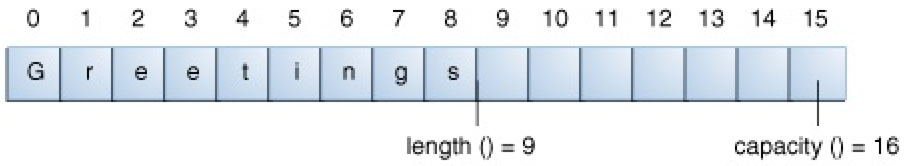
StringBuilder Constructors

Constructor	Description
`StringBuilder()`	Creates an empty string builder with a capacity of 16 (16 empty elements).
`StringBuilder(CharSequence cs)`	Constructs a string builder containing the same characters as the specified `CharSequence`, plus an extra 16 empty elements trailing the `CharSequence`.
`StringBuilder(int initCapacity)`	Creates an empty string builder with the specified initial capacity.
`StringBuilder(String s)`	Creates a string builder whose value is initialized by the specified string, plus an extra 16 empty elements trailing the string.

For example, the following code

```
// creates empty builder, capacity 16
StringBuilder sb = new StringBuilder();
// adds 9 character string at beginning
sb.append("Greetings");
```

will produce a string builder with a length of 9 and a capacity of 16:



The `StringBuilder` class has some methods related to length and capacity that the `String` class does not have:

Length and Capacity Methods

Method	Description
`void setLength(int newLength)`	Sets the length of the character sequence. If `newLength` is less than `length()`, the last characters in the character sequence are truncated. If `newLength` is greater than `length()`, null characters are added at the end of the character sequence.
`void ensureCapacity(int minCapacity)`	Ensures that the capacity is at least equal to the specified minimum.

A number of operations (for example, `append()`, `insert()`, or `setLength()`) can increase the length of the character sequence in the string builder so that the resultant `length()` would be greater than the current `capacity()`. When this happens, the capacity is automatically increased.

StringBuilder Operations

The principal operations on a `StringBuilder` that are not available in `String` are the `append()` and `insert()` methods, which are overloaded so as to accept data of any type. Each converts its argument to a string and then appends or inserts the characters of that string to the character sequence in the string builder. The append method always adds these characters at the end of the existing character sequence, while the insert method adds the characters at a specified point.

Here are a number of the methods of the `StringBuilder` class.

Various StringBuilder Methods

Method	Description

`StringBuilder append(boolean b)`
`StringBuilder append(char c)`
`StringBuilder append(char[] str)`
`StringBuilder append(char[] str, int offset, int len)`
	Appends the argument to this string builder. The

`StringBuilder append(double d)`
`StringBuilder append(float f)`
`StringBuilder append(int i)`
`StringBuilder append(long lng)`
`StringBuilder append(Object obj)`
`StringBuilder append(String s)`
	data is converted to a string before the append operation takes place.

`StringBuilder delete(int start, int end)`
`StringBuilder deleteCharAt(int index)`
	The first method deletes the subsequence from start to end-1 (inclusive) in the `StringBuilder`'s char sequence. The second method deletes the character located at `index`.

`StringBuilder insert(int offset, boolean b)`
`StringBuilder insert(int offset, char c)`
`StringBuilder insert(int offset, char[] str)`
`StringBuilder insert(int index, char[] str, int offset, int len)`
`StringBuilder insert(int offset, double d)`
`StringBuilder insert(int offset, float f)`
`StringBuilder insert(int offset, int i)`
`StringBuilder insert(int offset, long lng)`
`StringBuilder insert(int offset, Object obj)`
`StringBuilder insert(int offset, String s)`
	Inserts the second argument into the string builder. The first integer argument indicates the index before which the data is to be inserted. The data is converted to a string before the insert operation takes place.

`StringBuilder replace(int start, int end, String s)`
`void setCharAt(int index, char c)`
	Replaces the specified character(s) in this string builder.

`StringBuilder reverse()`	Reverses the sequence of characters in this string builder.
`String toString()`	Returns a string that contains the character sequence in the builder.

Note: You can use any `String` method on a `StringBuilder` object by first converting the string builder to a string with the `toString()` method of the `StringBuilder` class. Then convert the string back into a string builder using the `StringBuilder(String str)` constructor.

An Example

The `StringDemo` program that was listed in the section titled "Strings" is an example of a program that would be more efficient if a `StringBuilder` were used instead of a `String`.

`StringDemo` reversed a palindrome. Here, once again, is its listing:

```
public class StringDemo {
    public static void main(String[] args) {
        String palindrome = "Dot saw I was Tod";
        int len = palindrome.length();
        char[] tempCharArray = new char[len];
        char[] charArray = new char[len];

        // put original string in an
        // array of chars
        for (int i = 0; i < len; i++) {
            tempCharArray[i] =
                palindrome.charAt(i);
        }

        // reverse array of chars
        for (int j = 0; j < len; j++) {
            charArray[j] =
                tempCharArray[len - 1 - j];
        }

        String reversePalindrome =
            new String(charArray);
        System.out.println(reversePalindrome);
    }
}
```

Running the program produces this output:

```
doT saw I was toD
```

To accomplish the string reversal, the program converts the string to an array of characters (first `for` loop), reverses the array into a second array (second `for` loop), and then converts back to a string.

If you convert the `palindrome` string to a string builder, you can use the `reverse()` method in the `StringBuilder` class. It makes the code simpler and easier to read:

```
public class StringBuilderDemo {
    public static void main(String[] args) {
        String palindrome = "Dot saw I was Tod";

        StringBuilder sb = new StringBuilder(palindrome);

        sb.reverse();  // reverse it

        System.out.println(sb);
    }
}
```

Running this program produces the same output:

```
doT saw I was toD
```

Note that `println()` prints a string builder, as in:

```
System.out.println(sb);
```

because `sb.toString()` is called implicitly, as it is with any other object in a `println()` invocation.

Note: There is also a `StringBuffer` class that is *exactly* the same as the `StringBuilder` class, except that it is thread-safe by virtue of having its methods synchronized. Threads will be discussed in the lesson on concurrency.

Summary of Characters and Strings

Most of the time, if you are using a single character value, you will use the primitive `char` type. There are times, however, when you need to use a char as an object—for example, as a method argument where an object is expected. The Java programming language provides a *wrapper* class that "wraps" the `char` in a `Character` object for this purpose. An object of type `Character` contains a single field whose type is `char`. This `Character` class also offers a number of useful class (i.e., static) methods for manipulating characters.

Strings are a sequence of characters and are widely used in Java programming. In the Java programming language, strings are objects. The `String` class has over 60 methods and 13 constructors.

Most commonly, you create a string with a statement like

```
String s = "Hello world!";
```

rather than using one of the `String` constructors.

The `String` class has many methods to find and retrieve substrings; these can then be easily reassembled into new strings using the + concatenation operator.

The `String` class also includes a number of utility methods, among them `split()`, `toLowerCase()`, `toUpperCase()`, and `valueOf()`. The latter method is indispensable in converting user input strings to numbers. The `Number` subclasses also have methods for converting strings to numbers and vice versa.

In addition to the `String` class, there is also a `StringBuilder` class. Working with `StringBuilder` objects can sometimes be more efficient than working with strings. The `StringBuilder` class offers a few methods that can be useful for strings, among them `reverse()`. In general, however, the `String` class has a wider variety of methods.

A string can be converted to a string builder using a `StringBuilder` constructor. A string builder can be converted to a string with the `toString()` method.

Autoboxing and Unboxing

Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an `int` to an `Integer`, a `double` to a `Double`, and so on. If the conversion goes the other way, this is called *unboxing*.

Here is the simplest example of autoboxing:

```
Character ch = 'a';
```

The rest of the examples in this section use generics. If you are not yet familiar with the syntax of generics, see the Generics (Updated) lesson.

Consider the following code:

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2)
    li.add(i);
```

Although you add the `int` values as primitive types, rather than `Integer` objects, to `li`, the code compiles. Because `li` is a list of `Integer` objects, not a list of `int` values, you may wonder why the Java compiler does not issue a compile-time error. The compiler does not generate an error because it creates an `Integer` object from `i` and adds the object to `li`. Thus, the compiler converts the previous code to the following at runtime:

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2)
    li.add(Integer.valueOf(i));
```

Converting a primitive value (an `int`, for example) into an object of the corresponding wrapper class (`Integer`) is called autoboxing. The Java compiler applies autoboxing when a primitive value is:

- Passed as a parameter to a method that expects an object of the corresponding wrapper class.
- Assigned to a variable of the corresponding wrapper class.

Consider the following method:

```
public static int sumEven(List<Integer> li) {
    int sum = 0;
    for (Integer i: li)
        if (i % 2 == 0)
            sum += i;
        return sum;
}
```

Because the remainder (`%`) and unary plus (`+=`) operators do not apply to `Integer` objects, you may wonder why the Java compiler compiles the method without issuing any errors. The compiler does not generate an error because it invokes the `intValue` method to convert an `Integer` to an `int` at runtime:

```
public static int sumEven(List<Integer> li) {
    int sum = 0;
    for (Integer i : li)
        if (i.intValue() % 2 == 0)
            sum += i.intValue();
        return sum;
}
```

Converting an object of a wrapper type (`Integer`) to its corresponding primitive (`int`) value is called unboxing. The Java compiler applies unboxing when an object of a wrapper class is:

- Passed as a parameter to a method that expects a value of the corresponding primitive type.
- Assigned to a variable of the corresponding primitive type.

The `Unboxing` example shows how this works:

```
import java.util.ArrayList;
import java.util.List;

public class Unboxing {

    public static void main(String[] args) {
        Integer i = new Integer(-8);

        // 1. Unboxing through method invocation
        int absVal = absoluteValue(i);
        System.out.println("absolute value of " + i + " = " + absVal);

        List<Double> ld = new ArrayList<>();
        ld.add(3.1416);    //  is autoboxed through method invocation.

        // 2. Unboxing through assignment
        double pi = ld.get(0);
        System.out.println("pi = " + pi);
    }

    public static int absoluteValue(int i) {
        return (i < 0) ? -i : i;
    }
}
```

The program prints the following:

```
absolute value of -8 = 8
pi = 3.1416
```

Autoboxing and unboxing lets developers write cleaner code, making it easier to read. The following table lists the primitive types and their corresponding wrapper classes, which are used by the Java compiler for autoboxing and unboxing:

Primitive type	**Wrapper class**
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

Questions and Exercises: Characters and Strings

Questions

1. What is the initial capacity of the following string builder?

```
StringBuilder sb = new StringBuilder("Able was I ere I saw Elba.");
```

2. Consider the following string:

```
String hannah = "Did Hannah see bees? Hannah did.";
```

a. What is the value displayed by the expression `hannah.length()`?
b. What is the value returned by the method call `hannah.charAt(12)`?
c. Write an expression that refers to the letter `b` in the string referred to by `hannah`.
3. How long is the string returned by the following expression? What is the string?

```
"Was it a car or a cat I saw?".substring(9, 12)
```

4. In the following program, called `ComputeResult`, what is the value of `result` after each numbered line executes?

```
public class ComputeResult {
    public static void main(String[] args) {
        String original = "software";
        StringBuilder result = new StringBuilder("hi");
        int index = original.indexOf('a');

/*1*/   result.setCharAt(0, original.charAt(0));
/*2*/   result.setCharAt(1, original.charAt(original.length()-1));
/*3*/   result.insert(1, original.charAt(4));
/*4*/   result.append(original.substring(1,4));
/*5*/   result.insert(3, (original.substring(index, index+2) + " "));

        System.out.println(result);
    }
}
```

Exercises

1. Show two ways to concatenate the following two strings together to get the string `"Hi, mom."`:

```
String hi = "Hi, ";
String mom = "mom.";
```

2. Write a program that computes your initials from your full name and displays them.
3. An anagram is a word or a phrase made by transposing the letters of another word or phrase; for example, "parliament" is an anagram of "partial men," and "software" is an anagram of "swear oft." Write a program that figures out whether one string is an anagram of another string. The program should ignore white space and punctuation.

Check your answers.

Questions

1. Question: What is the initial capacity of the following string builder?

```
StringBuilder sb = new StringBuilder("Able was I ere I saw Elba.");
```

Answer: It's the length of the initial string + 16: 26 + 16 = 42.

2. Consider the following string:

```
String hannah = "Did Hannah see bees? Hannah did.";
```

a. Question: What is the value displayed by the expression `hannah.length()`?

Answer: 32.

b. Question: What is the value returned by the method call `hannah.charAt(12)`?

Answer: `e`.

c. Question: Write an expression that refers to the letter `b` in the string referred to by `hannah`.

Answer: `hannah.charAt(15)`.

3. Question: How long is the string returned by the following expression? What is the string?

```
"Was it a car or a cat I saw?".substring(9, 12)
```

Answer: It's 3 characters in length: `car`. It does not include the space after car.

4. Question: In the following program, called `ComputeResult`, what is the value of `result` after each numbered line executes?

```
public class ComputeResult {
    public static void main(String[] args) {
        String original = "software";
        StringBuilder result = new StringBuilder("hi");
        int index = original.indexOf('a');

/*1*/   result.setCharAt(0, original.charAt(0));
/*2*/   result.setCharAt(1, original.charAt(original.length()-1));
/*3*/   result.insert(1, original.charAt(4));
/*4*/   result.append(original.substring(1,4));
/*5*/   result.insert(3, (original.substring(index, index+2) + " "));

        System.out.println(result);
    }
}
```

Answer:

a. si

b. se

c. swe

d. sweoft

e. swear oft

Exercises

1. Exercise: Show two ways to concatenate the following two strings together to get the string `"Hi, mom."`:

```
String hi = "Hi, ";
String mom = "mom.";
```

Answer: `hi.concat(mom)` and `hi + mom`.

2. Exercise: Write a program that computes your initials from your full name and displays them.

Answer: `ComputeInitials`

```
public class ComputeInitials {
    public static void main(String[] args) {
        String myName = "Fred F. Flintstone";
        StringBuffer myInitials = new StringBuffer();
        int length = myName.length();

        for (int i = 0; i < length; i++) {
            if (Character.isUpperCase(myName.charAt(i))) {
                myInitials.append(myName.charAt(i));
            }
        }
        System.out.println("My initials are: " + myInitials);
    }
}
```

3. Exercise: An anagram is a word or a phrase made by transposing the letters of another word or phrase; for example, "parliament" is an anagram of "partial men," and "software" is an anagram of "swear oft." Write a program that figures out whether one string is an anagram of another string. The program should ignore white space and punctuation.

Answer: `Anagram`

```
public class Anagram {

    public static boolean areAnagrams(String string1,
                                      String string2) {

        String workingCopy1 = removeJunk(string1);
        String workingCopy2 = removeJunk(string2);

            workingCopy1 = workingCopy1.toLowerCase();
            workingCopy2 = workingCopy2.toLowerCase();

            workingCopy1 = sort(workingCopy1);
            workingCopy2 = sort(workingCopy2);

        return workingCopy1.equals(workingCopy2);
    }

    protected static String removeJunk(String string) {
        int i, len = string.length();
        StringBuilder dest = new StringBuilder(len);
                char c;

            for (i = (len - 1); i >= 0; i--) {
                c = string.charAt(i);
                if (Character.isLetter(c)) {
                        dest.append(c);
                }
            }

        return dest.toString();
    }

    protected static String sort(String string) {
            char[] charArray = string.toCharArray();

            java.util.Arrays.sort(charArray);
        return new String(charArray);
    }

    public static void main(String[] args) {
        String string1 = "Cosmo and Laine:";
        String string2 = "Maid, clean soon!";

        System.out.println();
        System.out.println("Testing whether the following "
                         + "strings are anagrams:");
        System.out.println("    String 1: " + string1);
        System.out.println("    String 2: " + string2);
        System.out.println();

        if (areAnagrams(string1, string2)) {
            System.out.println("They ARE anagrams!");
        } else {
            System.out.println("They are NOT anagrams!");
        }
        System.out.println();
    }
}
```

Lesson: Generics (Updated)

In any nontrivial software project, bugs are simply a fact of life. Careful planning, programming, and testing can help reduce their pervasiveness, but somehow, somewhere, they'll always find a way to creep into your code. This becomes especially apparent as new features are introduced and your code base grows in size and complexity.

Fortunately, some bugs are easier to detect than others. Compile-time bugs, for example, can be detected early on; you can use the compiler's error messages to figure out what the problem is and fix it, right then and there. Runtime bugs, however, can be much more problematic; they don't always surface immediately, and when they do, it may be at a point in the program that is far removed from the actual cause of the problem.

Generics add stability to your code by making more of your bugs detectable at compile time. After completing this lesson, you may want to follow up with the Generics tutorial by Gilad Bracha.

Note: See online version of topics in this ebook to download complete source code.

Why Use Generics?

In a nutshell, generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar *formal parameters* used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

Code that uses generics has many benefits over non-generic code:

- Stronger type checks at compile time.
  A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

- Elimination of casts.
  The following code snippet without generics requires casting:

  ```
  List list = new ArrayList();
  list.add("hello");
  String s = (String) list.get(0);
  ```

  When re-written to use generics, the code does not require casting:

  ```
  List<String> list = new ArrayList<String>();
  list.add("hello");
  String s = list.get(0);   // no cast
  ```
- Enabling programmers to implement generic algorithms.
  By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

Generic Types

A *generic type* is a generic class or interface that is parameterized over types. The following `Box` class will be modified to demonstrate the concept.

A Simple Box Class

Begin by examining a non-generic `Box` class that operates on objects of any type. It needs only to provide two methods: `set`, which adds an object to the box, and `get`, which retrieves it:

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

Since its methods accept or return an `Object`, you are free to pass in whatever you want, provided that it is not one of the primitive types. There is no way to verify, at compile time, how the class is used. One part of the code may place an `Integer` in the box and expect to get `Integer`s out of it, while another part of the code may mistakenly pass in a `String`, resulting in a runtime error.

A Generic Version of the Box Class

A *generic class* is defined with the following format:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the *type parameters* (also called *type variables*) `T1`, `T2`, ..., and `Tn`.

To update the `Box` class to use generics, you create a *generic type declaration* by changing the code "`public class Box`" to "`public class Box<T>`". This introduces the type variable, `T`, that can be used anywhere inside the class.

With this change, the `Box` class becomes:

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

As you can see, all occurrences of `Object` are replaced by `T`. A type variable can be any **non-primitive** type you specify: any class type, any interface type, any array type, or even another type variable.

This same technique can be applied to create generic interfaces.

Type Parameter Naming Conventions

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable naming conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

You'll see these names used throughout the Java SE API and the rest of this lesson.

Invoking and Instantiating a Generic Type

To reference the generic `Box` class from within your code, you must perform a *generic type invocation*, which replaces `T` with some concrete value, such as `Integer`:

```
Box<Integer> integerBox;
```

You can think of a generic type invocation as being similar to an ordinary method invocation, but instead of passing an argument to a method, you are passing a *type argument* `Integer` in this case to the `Box` class itself.

Type Parameter and Type Argument Terminology: Many developers use the terms "type parameter" and "type argument" interchangeably, but these terms are not the same. When coding, one provides type arguments in order to create a parameterized type. Therefore, the `T` in `Foo<T>` is a type parameter and the `String` in `Foo<String> f` is a type argument. This lesson observes this definition when using these terms.

Like any other variable declaration, this code does not actually create a new `Box` object. It simply declares that `integerBox` will hold a reference to a "`Box` of `Integer`", which is how `Box<Integer>` is read.

An invocation of a generic type is generally known as a *parameterized type*.

To instantiate this class, use the `new` keyword, as usual, but place `<Integer>` between the class name and the parenthesis:

```
Box<Integer> integerBox = new Box<Integer>();
```

The Diamond

In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, <>, is informally called *the diamond*. For example, you can create an instance of `Box<Integer>` with the following statement:

```
Box<Integer> integerBox = new Box<>();
```

For more information on diamond notation and type inference, see Type Inference.

Multiple Type Parameters

As mentioned previously, a generic class can have multiple type parameters. For example, the generic `OrderedPair` class, which implements the generic `Pair` interface:

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

The following statements create two instantiations of the `OrderedPair` class:

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");
```

The code, `new OrderedPair<String, Integer>`, instantiates `K` as a `String` and `V` as an `Integer`. Therefore, the parameter types of `OrderedPair`'s constructor are `String` and `Integer`, respectively. Due to autoboxing, it is valid to pass a `String` and an `int` to the class.

As mentioned in The Diamond, because a Java compiler can infer the `K` and `V` types from the declaration `OrderedPair<String, Integer>`, these statements can be shortened using diamond notation:

```
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
OrderedPair<String, String>  p2 = new OrderedPair<>("hello", "world");
```

To create a generic interface, follow the same conventions as for creating a generic class.

Parameterized Types

You can also substitute a type parameter (i.e., `K` or `V`) with a parameterized type (i.e., `List<String>`). For example, using the `OrderedPair<K, V>` example:

```
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new Box<Integer>(...));
```

Raw Types

A *raw type* is the name of a generic class or interface without any type arguments. For example, given the generic `Box` class:

```
public class Box<T> {
    public void set(T t) { /* ... */ }
    // ...
}
```

To create a parameterized type of `Box<T>`, you supply an actual type argument for the formal type parameter `T`:

```
Box<Integer> intBox = new Box<>();
```

If the actual type argument is omitted, you create a raw type of `Box<T>`:

```
Box rawBox = new Box();
```

Therefore, `Box` is the raw type of the generic type `Box<T>`. However, a non-generic class or interface type is *not* a raw type.

Raw types show up in legacy code because lots of API classes (such as the `Collections` classes) were not generic prior to JDK 5.0. When using raw types, you essentially get pre-generics behavior a `Box` gives you `Object`s. For backward compatibility, assigning a parameterized type to its raw type is allowed:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;               // OK
```

But if you assign a raw type to a parameterized type, you get a warning:

```
Box rawBox = new Box();           // rawBox is a raw type of Box<T>
Box<Integer> intBox = rawBox;     // warning: unchecked conversion
```

You also get a warning if you use a raw type to invoke generic methods defined in the corresponding generic type:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;
rawBox.set(8);  // warning: unchecked invocation to set(T)
```

The warning shows that raw types bypass generic type checks, deferring the catch of unsafe code to runtime. Therefore, you should avoid using raw types.

The Type Erasure section has more information on how the Java compiler uses raw types.

Unchecked Error Messages

As mentioned previously, when mixing legacy code with generic code, you may encounter warning messages similar to the following:

```
Note: Example.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

This can happen when using an older API that operates on raw types, as shown in the following example:

```
public class WarningDemo {
    public static void main(String[] args){
        Box<Integer> bi;
        bi = createBox();
    }

    static Box createBox(){
        return new Box();
    }
}
```

The term "unchecked" means that the compiler does not have enough type information to perform all type checks necessary to ensure type safety. The "unchecked" warning is disabled, by default, though the compiler gives a hint. To see all "unchecked" warnings, recompile with `-Xlint:unchecked`.

Recompiling the previous example with `-Xlint:unchecked` reveals the following additional information:

```
WarningDemo.java:4: warning: [unchecked] unchecked conversion
found   : Box
required: Box<java.lang.Integer>
        bi = createBox();
                      ^
1 warning
```

To completely disable unchecked warnings, use the `-Xlint:-unchecked` flag. The `@SuppressWarnings("unchecked")` annotation suppresses unchecked warnings. If you are unfamiliar with the `@SuppressWarnings` syntax, see Annotations.

Generic Methods

Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed, as well as generic class constructors.

The syntax for a generic method includes a type parameter, inside angle brackets, and appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type.

The `Util` class includes a generic method, `compare`, which compares two `Pair` objects:

```
public class Util {
    // Generic static method
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}

public class Pair<K, V> {

    private K key;
    private V value;

    // Generic constructor
    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    // Generic methods
    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

The complete syntax for invoking this method would be:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

The type has been explicitly provided, as shown in bold. Generally, this can be left out and the compiler will infer the type that is needed:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.compare(p1, p2);
```

This feature, known as *type inference*, allows you to invoke a generic method as an ordinary method, without specifying a type between angle brackets. This topic is further discussed in the following section, Type Inference.

Bounded Type Parameters

There may be times when you want to restrict the types that can be used as type arguments in a parameterized type. For example, a method that operates on numbers might only want to accept instances of `Number` or its subclasses. This is what *bounded type parameters* are for.

To declare a bounded type parameter, list the type parameter's name, followed by the `extends` keyword, followed by its *upper bound*, which in this example is `Number`. Note that, in this context, `extends` is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

```
public class Box<T> {

    private T t;

    public void set(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(10));
        integerBox.inspect("some text"); // error: this is still String!
    }
}
```

By modifying our generic method to include this bounded type parameter, compilation will now fail, since our invocation of `inspect` still includes a `String`:

```
Box.java:21: <U>inspect(U) in Box<java.lang.Integer> cannot
  be applied to (java.lang.String)
                        integerBox.inspect("10");
                                  ^
1 error
```

In addition to limiting the types you can use to instantiate a generic type, bounded type parameters allow you to invoke methods defined in the bounds:

```
public class NaturalNumber<T extends Integer> {

    private T n;

    public NaturalNumber(T n)  { this.n = n; }

    public boolean isEven() {
        return n.intValue() % 2 == 0;
    }
    // ...
}
```

The `isEven` method invokes the `intValue` method defined in the `Integer` class through `n`.

Multiple Bounds

The preceding example illustrates the use of a type parameter with a single bound, but a type parameter can have *multiple bounds*:

```
<T extends B1 & B2 & B3>
```

A type variable with multiple bounds is a subtype of all the types listed in the bound. If one of the bounds is a class, it must be specified first. For example:

```
Class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
```

If bound `A` is not specified first, you get a compile-time error:

```
class D <T extends B & A & C> { /* ... */ }  // compile-time error
```

Generic Methods and Bounded Type Parameters

Bounded type parameters are key to the implementation of generic algorithms. Consider the following method that counts the number of elements in an array `T[]` that are greater than a specified element `elem`.

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem)  // compiler error
            ++count;
    return count;
}
```

The implementation of the method is straightforward, but it does not compile because the greater than operator (>) applies only to primitive types such as `short`, `int`, `double`, `long`, `float`, `byte`, and `char`. You cannot use the > operator to compare objects. To fix the problem, use a type parameter bounded by the `Comparable<T>` interface:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

The resulting code will be:

```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

Generics, Inheritance, and Subtypes

As you already know, it is possible to assign an object of one type to an object of another type provided that the types are compatible. For example, you can assign an `Integer` to an `Object`, since `Object` is one of `Integer`'s supertypes:

```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger;   // OK
```

In object-oriented terminology, this is called an "is a" relationship. Since an `Integer` *is a* kind of `Object`, the assignment is allowed. But `Integer` is also a kind of `Number`, so the following code is valid as well:

```
public void someMethod(Number n) { /* ... */ }

someMethod(new Integer(10));   // OK
someMethod(new Double(10.1));   // OK
```

The same is also true with generics. You can perform a generic type invocation, passing `Number` as its type argument, and any subsequent invocation of `add` will be allowed if the argument is compatible with `Number`:

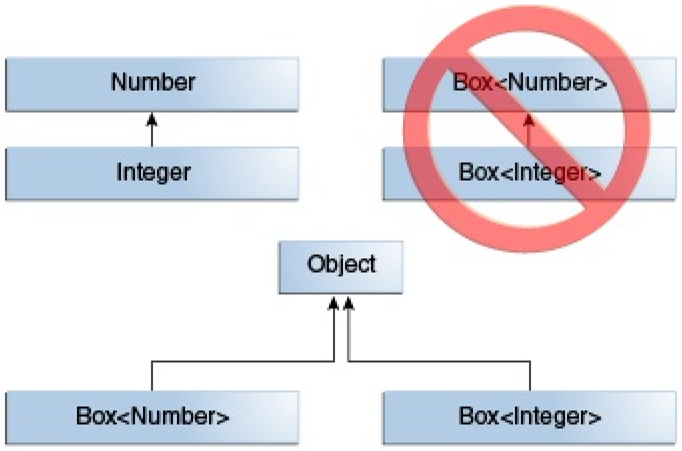
```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));   // OK
box.add(new Double(10.1));  // OK
```

Now consider the following method:

```
public void boxTest(Box<Number> n) { /* ... */ }
```

What type of argument does it accept? By looking at its signature, you can see that it accepts a single argument whose type is `Box<Number>`. But what does that mean? Are you allowed to pass in `Box<Integer>` or `Box<Double>`, as you might expect? The answer is "no", because `Box<Integer>` and `Box<Double>` are not subtypes of `Box<Number>`.

This is a common misunderstanding when it comes to programming with generics, but it is an important concept to learn.



`Box<Integer>` is not a subtype of `Box<Number>` even though `Integer` is a subtype of `Number`.

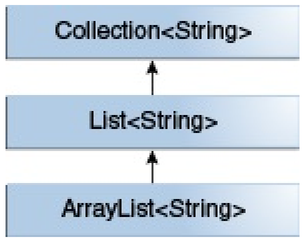
Note: Given two concrete types `A` and `B` (for example, `Number` and `Integer`), `MyClass<A>` has no relationship to `MyClass<B>`, regardless of whether or not `A` and `B` are related. The common parent of `MyClass<A>` and `MyClass<B>` is `Object`.

For information on how to create a subtype-like relationship between two generic classes when the type parameters are related, see Wildcards and Subtyping.

Generic Classes and Subtyping

You can subtype a generic class or interface by extending or implementing it. The relationship between the type parameters of one class or interface and the type parameters of another are determined by the `extends` and `implements` clauses.

Using the `Collections` classes as an example, `ArrayList<E>` implements `List<E>`, and `List<E> extends Collection<E>`. So `ArrayList<String>` is a subtype of `List<String>`, which is a subtype of `Collection<String>`. So long as you do not vary the type argument, the subtyping relationship is preserved between the types.



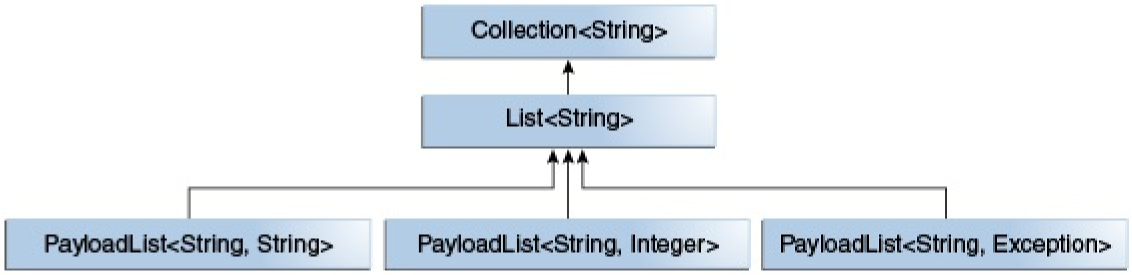
A sample `Collections` hierarchy

Now imagine we want to define our own list interface, `PayloadList`, that associates an optional value of generic type `P` with each element. Its declaration might look like:

```
interface PayloadList<E,P> extends List<E> {
  void setPayload(int index, P val);
  ...
}
```

The following parameterizations of `PayloadList` are subtypes of `List<String>`:

- `PayloadList<String,String>`
- `PayloadList<String,Integer>`
- `PayloadList<String,Exception>`


Collection<String>
List<String>
PayloadList<String, String>
PayloadList<String, Integer>
PayloadList<String, Exception>

A sample `PayloadList` hierarchy

Type Inference

Type inference is a Java compiler's ability to look at each method invocation and corresponding declaration to determine the type argument (or arguments) that make the invocation applicable. The inference algorithm determines the types of the arguments and, if available, the type that the result is being assigned, or returned. Finally, the inference algorithm tries to find the *most specific* type that works with all of the arguments.

To illustrate this last point, in the following example, inference determines that the second argument being passed to the `pick` method is of type `Serializable`:

```
static <T> T pick(T a1, T a2) { return a2; }
Serializable s = pick("d", new ArrayList<String>());
```

Type Inference and Generic Methods

Generic Methods introduced you to type inference, which enables you to invoke a generic method as you would an ordinary method, without specifying a type between angle brackets. Consider the following example, `BoxDemo`, which requires the `Box` class:

```
public class BoxDemo {

  public static <U> void addBox(U u,
      java.util.List<Box<U>> boxes) {
    Box<U> box = new Box<>();
    box.set(u);
    boxes.add(box);
  }

  public static <U> void outputBoxes(java.util.List<Box<U>> boxes) {
    int counter = 0;
    for (Box<U> box: boxes) {
      U boxContents = box.get();
      System.out.println("Box #" + counter + " contains [" +
             boxContents.toString() + "]");
      counter++;
    }
  }

  public static void main(String[] args) {
    java.util.ArrayList<Box<Integer>> listOfIntegerBoxes =
      new java.util.ArrayList<>();
    BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
    BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);
    BoxDemo.addBox(Integer.valueOf(30), listOfIntegerBoxes);
    BoxDemo.outputBoxes(listOfIntegerBoxes);
  }
}
```

The following is the output from this example:

```
Box #0 contains [10]
Box #1 contains [20]
Box #2 contains [30]
```

The generic method `addBox` defines one type parameter named `U`. Generally, a Java compiler can infer the type parameters of a generic method call. Consequently, in most cases, you do not have to specify them. For example, to invoke the generic method `addBox`, you can specify the type parameter with a *type witness* as follows:

```
BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
```

Alternatively, if you omit the type witness,a Java compiler automatically infers (from the method's arguments) that the type parameter is `Integer`:

```
BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);
```

Type Inference and Instantiation of Generic Classes

You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type parameters (<>) as long as the compiler can infer the type arguments from the context. This pair of angle brackets is informally called the diamond.

For example, consider the following variable declaration:

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

You can substitute the parameterized type of the constructor with an empty set of type parameters (<>):

```
Map<String, List<String>> myMap = new HashMap<>();
```

Note that to take advantage of type inference during generic class instantiation, you must use the diamond. In the following example, the compiler generates an unchecked conversion warning because the `HashMap()` constructor refers to the `HashMap` raw type, not the `Map<String, List<String>>` type:

```
Map<String, List<String>> myMap = new HashMap(); // unchecked conversion warning
```

Type Inference and Generic Constructors of Generic and Non-Generic Classes

Note that constructors can be generic (in other words, declare their own formal type parameters) in both generic and non-generic classes. Consider the following example:

```
class MyClass<X> {
  <T> MyClass(T t) {
    // ...
  }
}
```

Consider the following instantiation of the class `MyClass`:

```
new MyClass<Integer>("")
```

This statement creates an instance of the parameterized type `MyClass<Integer>`; the statement explicitly specifies the type `Integer` for the formal type parameter, `X`, of the generic class `MyClass<X>`. Note that the constructor for this generic class contains a formal type parameter, `T`. The compiler infers the type `String` for the formal type parameter, `T`, of the constructor of this generic class (because the actual parameter of this constructor is a `String` object).

Compilers from releases prior to Java SE 7 are able to infer the actual type parameters of generic constructors, similar to generic methods. However, compilers in Java SE 7 and later can infer the actual type parameters of the generic class being instantiated if you use the diamond (`<>`). Consider the following example:

```
MyClass<Integer> myObject = new MyClass<>("");
```

In this example, the compiler infers the type `Integer` for the formal type parameter, `X`, of the generic class `MyClass<X>`. It infers the type `String` for the formal type parameter, `T`, of the constructor of this generic class.

Note: It is important to note that the inference algorithm uses only invocation arguments, target types, and possibly an obvious expected return type to infer types. The inference algorithm does not use results from later in the program.

Target Types

The Java compiler takes advantage of target typing to infer the type parameters of a generic method invocation. The *target type* of an expression is the data type that the Java compiler expects depending on where the expression appears. Consider the method `Collections.emptyList`, which is declared as follows:

```
static <T> List<T> emptyList();
```

Consider the following assignment statement:

```
List<String> listOne = Collections.emptyList();
```

This statement is expecting an instance of `List<String>`; this data type is the target type. Because the method `emptyList` returns a value of type `List<T>`, the compiler infers that the type argument `T` must be the value `String`. This works in both Java SE 7 and 8. Alternatively, you could use a type witness and specify the value of `T` as follows:

```
List<String> listOne = Collections.<String>emptyList();
```

However, this is not necessary in this context. It was necessary in other contexts, though. Consider the following method:

```
void processStringList(List<String> stringList) {
    // process stringList
}
```

Suppose you want to invoke the method `processStringList` with an empty list. In Java SE 7, the following statement does not compile:

```
processStringList(Collections.emptyList());
```

The Java SE 7 compiler generates an error message similar to the following:

```
List<Object> cannot be converted to List<String>
```

The compiler requires a value for the type argument `T` so it starts with the value `Object`. Consequently, the invocation of `Collections.emptyList` returns a value of type `List<Object>`, which is incompatible with the method `processStringList`. Thus, in Java SE 7, you must specify the value of the value of the type argument as follows:

```
processStringList(Collections.<String>emptyList());
```

This is no longer necessary in Java SE 8. The notion of what is a target type has been expanded to include method arguments, such as the argument to the method `processStringList`. In this case, `processStringList` requires an argument of type `List<String>`. The method `Collections.emptyList` returns a value of `List<T>`, so using the target type of `List<String>`, the compiler infers that the type argument `T` has a value of `String`. Thus, in Java SE 8, the following statement compiles:

```
processStringList(Collections.emptyList());
```

See Target Typing in Lambda Expressions for more information.

Wildcards

In generic code, the question mark (`?`), called the *wildcard*, represents an unknown type. The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type (though it is better programming practice to be more specific). The wildcard is never used as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

The following sections discuss wildcards in more detail, including upper bounded wildcards, lower bounded wildcards, and wildcard capture.

Upper Bounded Wildcards

You can use an upper bounded wildcard to relax the restrictions on a variable. For example, say you want to write a method that works on `List<Integer>`, `List<Double>`, *and* `List<Number>`; you can achieve this by using an upper bounded wildcard.

To declare an upper-bounded wildcard, use the wildcard character ('`?`'), followed by the `extends` keyword, followed by its *upper bound*. Note that, in this context, `extends` is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

To write the method that works on lists of `Number` and the subtypes of `Number`, such as `Integer`, `Double`, and `Float`, you would specify `List<? extends Number>`. The term `List<Number>` is more restrictive than `List<? extends Number>` because the former matches a list of type `Number` only, whereas the latter matches a list of type `Number` or any of its subclasses.

Consider the following `process` method:

```
public static void process(List<? extends Foo> list) { /* ... */ }
```

The upper bounded wildcard, `<? extends Foo>`, where `Foo` is any type, matches `Foo` and any subtype of `Foo`. The `process` method can access the list elements as type `Foo`:

```
public static void process(List<? extends Foo> list) {
    for (Foo elem : list) {
        // ...
    }
}
```

In the `foreach` clause, the `elem` variable iterates over each element in the list. Any method defined in the `Foo` class can now be used on `elem`.

The `sumOfList` method returns the sum of the numbers in a list:

```
public static double sumOfList(List<? extends Number> list) {
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}
```

The following code, using a list of `Integer` objects, prints `sum = 6.0`:

```
List<Integer> li = Arrays.asList(1, 2, 3);
System.out.println("sum = " + sumOfList(li));
```

A list of `Double` values can use the same `sumOfList` method. The following code prints `sum = 7.0`:

```
List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);
System.out.println("sum = " + sumOfList(ld));
```

Unbounded Wildcards

The unbounded wildcard type is specified using the wildcard character (`?`), for example, `List<?>`. This is called a *list of unknown type*. There are two scenarios where an unbounded wildcard is a useful approach:

- If you are writing a method that can be implemented using functionality provided in the `Object` class.
- When the code is using methods in the generic class that don't depend on the type parameter. For example, `List.size` or `List.clear`. In fact, `Class<?>` is so often used because most of the methods in `Class<T>` do not depend on `T`.

Consider the following method, `printList`:

```
public static void printList(List<Object> list) {
    for (Object elem : list)
        System.out.println(elem + " ");
    System.out.println();
}
```

The goal of `printList` is to print a list of any type, but it fails to achieve that goal it prints only a list of `Object` instances; it cannot print `List<Integer>`, `List<String>`, `List<Double>`, and so on, because they are not subtypes of `List<Object>`. To write a generic `printList` method, use `List<?>`:

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

Because for any concrete type `A`, `List<A>` is a subtype of `List<?>`, you can use `printList` to print a list of any type:

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<String>  ls = Arrays.asList("one", "two", "three");
printList(li);
printList(ls);
```

Note: The `Arrays.asList` method is used in examples throughout this lesson. This static factory method converts the specified array and returns a fixed-size list.

It's important to note that `List<Object>` and `List<?>` are not the same. You can insert an `Object`, or any subtype of `Object`, into a `List<Object>`. But you can only insert `null` into a `List<?>`. The Guidelines for Wildcard Use section has more information on how to determine what kind of wildcard, if any, should be used in a given situation.

Lower Bounded Wildcards

The Upper Bounded Wildcards section shows that an upper bounded wildcard restricts the unknown type to be a specific type or a subtype of that type and is represented using the `extends` keyword. In a similar way, a *lower bounded* wildcard restricts the unknown type to be a specific type or a *super type* of that type.

A lower bounded wildcard is expressed using the wildcard character ('`?`'), following by the `super` keyword, followed by its *lower bound*: `<? super A>`.

Note: You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

Say you want to write a method that puts `Integer` objects into a list. To maximize flexibility, you would like the method to work on `List<Integer>`, `List<Number>`, and `List<Object>` anything that can hold `Integer` values.

To write the method that works on lists of `Integer` and the supertypes of `Integer`, such as `Integer`, `Number`, and `Object`, you would specify `List<? super Integer>`. The term `List<Integer>` is more restrictive than `List<? super Integer>` because the former matches a list of type `Integer` only, whereas the latter matches a list of any type that is a supertype of `Integer`.

The following code adds the numbers 1 through 10 to the end of a list:

```
public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```

The Guidelines for Wildcard Use section provides guidance on when to use upper bounded wildcards and when to use lower bounded wildcards.

Wildcards and Subtyping

As described in Generics, Inheritance, and Subtypes, generic classes or interfaces are not related merely because there is a relationship between their types. However, you can use wildcards to create a relationship between generic classes or interfaces.

Given the following two regular (non-generic) classes:

```
class A { /* ... */ }
class B extends A { /* ... */ }
```

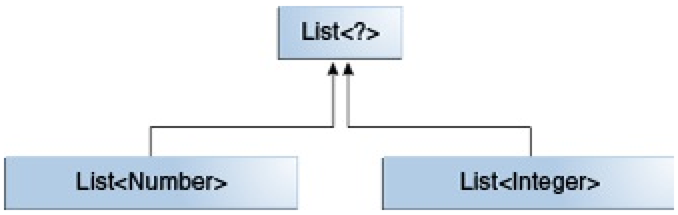
It would be reasonable to write the following code:

```
B b = new B();
A a = b;
```

This example shows that inheritance of regular classes follows this rule of subtyping: class `B` is a subtype of class `A` if `B` extends `A`. This rule does not apply to generic types:

```
List<B> lb = new ArrayList<>();
List<A> la = lb;   // compile-time error
```

Given that `Integer` is a subtype of `Number`, what is the relationship between `List<Integer>` and `List<Number>`?



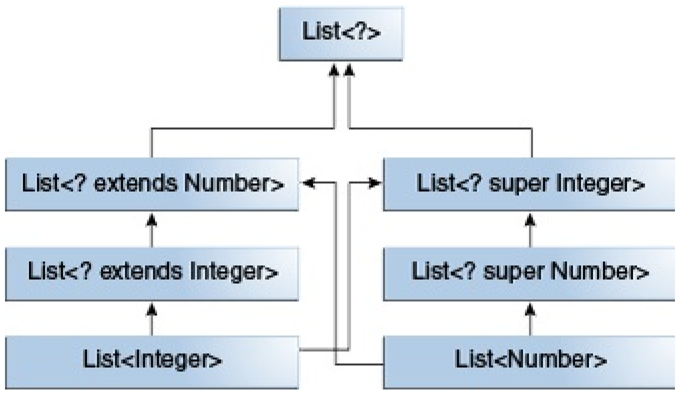
The common parent is `List<?>`.

Although `Integer` is a subtype of `Number`, `List<Integer>` is not a subtype of `List<Number>` and, in fact, these two types are not related. The common parent of `List<Number>` and `List<Integer>` is `List<?>`.

In order to create a relationship between these classes so that the code can access `Number`'s methods through `List<Integer>`'s elements, use an upper bounded wildcard:

```
List<? extends Integer> intList = new ArrayList<>();
List<? extends Number>  numList = intList;  // OK. List<? extends Integer> is a subtype of List<? extends Number>
```

Because `Integer` is a subtype of `Number`, and `numList` is a list of `Number` objects, a relationship now exists between `intList` (a list of `Integer` objects) and `numList`. The following diagram shows the relationships between several `List` classes declared with both upper and lower bounded wildcards.



A hierarchy of several generic `List` class declarations.

The Guidelines for Wildcard Use section has more information about the ramifications of using upper and lower bounded wildcards.

Wildcard Capture and Helper Methods

In some cases, the compiler infers the type of a wildcard. For example, a list may be defined as `List<?>` but, when evaluating an expression, the compiler infers a particular type from the code. This scenario is known as *wildcard capture*.

For the most part, you don't need to worry about wildcard capture, except when you see an error message that contains the phrase "capture of".

The `WildcardError` example produces a capture error when compiled:

```
import java.util.List;

public class WildcardError {

    void foo(List<?> i) {
        i.set(0, i.get(0));
    }
}
```

In this example, the compiler processes the `i` input parameter as being of type `Object`. When the `foo` method invokes List.set(int, E), the compiler is not able to confirm the type of object that is being inserted into the list, and an error is produced. When this type of error occurs it typically means that the compiler believes that you are assigning the wrong type to a variable. Generics were added to the Java language for this reason to enforce type safety at compile time.

The `WildcardError` example generates the following error when compiled by Oracle's JDK 7 `javac` implementation:

```
WildcardError.java:6: error: method set in interface List<E> cannot be applied to given types;
    i.set(0, i.get(0));
     ^
  required: int,CAP#1
  found: int,Object
  reason: actual argument Object cannot be converted to CAP#1 by method invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
1 error
```

In this example, the code is attempting to perform a safe operation, so how can you work around the compiler error? You can fix it by writing a *private helper method* which captures the wildcard. In this case, you can work around the problem by creating the private helper method, `fooHelper`, as shown in `WildcardFixed`:

```
public class WildcardFixed {

    void foo(List<?> i) {
        fooHelper(i);
    }
    // Helper method created so that the wildcard can be captured
    // through type inference.
    private <T> void fooHelper(List<T> l) {
        l.set(0, l.get(0));
    }

}
```

Thanks to the helper method, the compiler uses inference to determine that `T` is `CAP#1`, the capture variable, in the invocation. The example now compiles successfully.

By convention, helper methods are generally named *`originalMethodName`*`Helper`.

Now consider a more complex example, `WildcardErrorBad`:

```
import java.util.List;

public class WildcardErrorBad {

    void swapFirst(List<? extends Number> l1, List<? extends Number> l2) {
      Number temp = l1.get(0);
      l1.set(0, l2.get(0)); // expected a CAP#1 extends Number,
                            // got a CAP#2 extends Number;
                            // same bound, but different types
      l2.set(0, temp);      // expected a CAP#1 extends Number,
                            // got a Number
    }
}
```

In this example, the code is attempting an unsafe operation. For example, consider the following invocation of the `swapFirst` method:

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<Double>  ld = Arrays.asList(10.10, 20.20, 30.30);
swapFirst(li, ld);
```

While `List<Integer>` and `List<Double>` both fulfill the criteria of `List<? extends Number>`, it is clearly incorrect to take an item from a list of `Integer` values and attempt to place it into a list of `Double` values.

Compiling the code with Oracle's JDK `javac` compiler produces the following error:

```
WildcardErrorBad.java:7: error: method set in interface List<E> cannot be applied to given types;
      l1.set(0, l2.get(0)); // expected a CAP#1 extends Number,
        ^
  required: int,CAP#1
  found: int,Number
  reason: actual argument Number cannot be converted to CAP#1 by method invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Number from capture of ? extends Number
WildcardErrorBad.java:10: error: method set in interface List<E> cannot be applied to given types;
      l2.set(0, temp);      // expected a CAP#1 extends Number,
        ^
  required: int,CAP#1
  found: int,Number
  reason: actual argument Number cannot be converted to CAP#1 by method invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Number from capture of ? extends Number
WildcardErrorBad.java:15: error: method set in interface List<E> cannot be applied to given types;
        i.set(0, i.get(0));
         ^
  required: int,CAP#1
  found: int,Object
  reason: actual argument Object cannot be converted to CAP#1 by method invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
3 errors
```

There is no helper method to work around the problem, because the code is fundamentally wrong.

Guidelines for Wildcard Use

One of the more confusing aspects when learning to program with generics is determining when to use an upper bounded wildcard and when to use a lower bounded wildcard. This page provides some guidelines to follow when designing your code.

For purposes of this discussion, it is helpful to think of variables as providing one of two functions:

An "In" Variable
An "in" variable serves up data to the code. Imagine a copy method with two arguments: `copy(src, dest)`. The `src` argument provides the data to be copied, so it is the "in" parameter.

An "Out" Variable
An "out" variable holds data for use elsewhere. In the copy example, `copy(src, dest)`, the `dest` argument accepts data, so it is the "out" parameter.

Of course, some variables are used both for "in" and "out" purposes this scenario is also addressed in the guidelines.

You can use the "in" and "out" principle when deciding whether to use a wildcard and what type of wildcard is appropriate. The following list provides the guidelines to follow:

Wildcard Guidelines:

- An "in" variable is defined with an upper bounded wildcard, using the `extends` keyword.
- An "out" variable is defined with a lower bounded wildcard, using the `super` keyword.
- In the case where the "in" variable can be accessed using methods defined in the `Object` class, use an unbounded wildcard.
- In the case where the code needs to access the variable as both an "in" and an "out" variable, do not use a wildcard.

These guidelines do not apply to a method's return type. Using a wildcard as a return type should be avoided because it forces programmers using the code to deal with wildcards.

A list defined by `List<? extends ...>` can be informally thought of as read-only, but that is not a strict guarantee. Suppose you have the following two classes:

```
class NaturalNumber {

    private int i;

    public NaturalNumber(int i) { this.i = i; }
    // ...
}

class EvenNumber extends NaturalNumber {

    public EvenNumber(int i) { super(i); }
    // ...
}
```

Consider the following code:

```
List<EvenNumber> le = new ArrayList<>();
List<? extends NaturalNumber> ln = le;
ln.add(new NaturalNumber(35));  // compile-time error
```

Because `List<EvenNumber>` is a subtype of `List<? extends NaturalNumber>`, you can assign `le` to `ln`. But you cannot use `ln` to add a natural number to a list of even numbers. The following operations on the list are possible:

- You can add `null`.
- You can invoke `clear`.
- You can get the iterator and invoke `remove`.
- You can capture the wildcard and write elements that you've read from the list.

You can see that the list defined by `List<? extends NaturalNumber>` is not read-only in the strictest sense of the word, but you might think of it that way because you cannot store a new element or change an existing element in the list.

Type Erasure

Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming. To implement generics, the Java compiler applies type erasure to:

- Replace all type parameters in generic types with their bounds or `Object` if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
- Insert type casts if necessary to preserve type safety.
- Generate bridge methods to preserve polymorphism in extended generic types.

Type erasure ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead.

Erasure of Generic Types

During the type erasure process, the Java compiler erases all type parameters and replaces each with its first bound if the type parameter is bounded, or `Object` if the type parameter is unbounded.

Consider the following generic class that represents a node in a singly linked list:

```
public class Node<T> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) }
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

Because the type parameter `T` is unbounded, the Java compiler replaces it with `Object`:

```
public class Node {

    private Object data;
    private Node next;

    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Object getData() { return data; }
    // ...
}
```

In the following example, the generic `Node` class uses a bounded type parameter:

```
public class Node<T extends Comparable<T>> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

The Java compiler replaces the bounded type parameter `T` with the first bound class, `Comparable`:

```
public class Node {
    private Comparable data;
    private Node next;

    public Node(Comparable data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Comparable getData() { return data; }
    // ...
}
```

Erasure of Generic Methods

The Java compiler also erases type parameters in generic method arguments. Consider the following generic method:

```
// Counts the number of occurrences of elem in anArray.
//
public static <T> int count(T[] anArray, T elem) {
    int cnt = 0;
    for (T e : anArray)
        if (e.equals(elem))
            ++cnt;
        return cnt;
}
```

Because `T` is unbounded, the Java compiler replaces it with `Object`:

```
public static int count(Object[] anArray, Object elem) {
    int cnt = 0;
    for (Object e : anArray)
        if (e.equals(elem))
            ++cnt;
        return cnt;
}
```

Suppose the following classes are defined:

```
class Shape { /* ... */ }
class Circle extends Shape { /* ... */ }
class Rectangle extends Shape { /* ... */ }
```

You can write a generic method to draw different shapes:

```
public static <T extends Shape> void draw(T shape) { /* ... */ }
```

The Java compiler replaces `T` with `Shape`:

```
public static void draw(Shape shape) { /* ... */ }
```

Effects of Type Erasure and Bridge Methods

Sometimes type erasure causes a situation that you may not have anticipated. The following example shows how this can occur. The example (described in Bridge Methods) shows how a compiler sometimes creates a synthetic method, called a bridge method, as part of the type erasure process.

Given the following two classes:

```
public class Node<T> {

    private T data;

    public Node(T data) { this.data = data; }

    public void setData(T data) {
        System.out.println("Node.setData");
        this.data = data;
    }
}

public class MyNode extends Node<Integer> {
    public MyNode(Integer data) { super(data); }

    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }
}
```

Consider the following code:

```
MyNode mn = new MyNode(5);
Node n = mn;            // A raw type - compiler throws an unchecked warning
n.setData("Hello");     // Causes a ClassCastException to be thrown.
Integer x = mn.data;
```

After type erasure, this code becomes:

```
MyNode mn = new MyNode(5);
Node n = (MyNode)mn;         // A raw type - compiler throws an unchecked warning
n.setData("Hello");
Integer x = (String)mn.data; // Causes a ClassCastException to be thrown.
```

Here is what happens as the code is executed:

- `n.setData("Hello");` causes the method `setData(Object)` to be executed on the object of class `MyNode`. (The `MyNode` class inherited `setData(Object)` from `Node`.)
- In the body of `setData(Object)`, the data field of the object referenced by `n` is assigned to a `String`.
- The data field of that same object, referenced via `mn`, can be accessed and is expected to be an integer (since `mn` is a `MyNode` which is a `Node<Integer>`.
- Trying to assign a `String` to an `Integer` causes a `ClassCastException` from a cast inserted at

the assignment by a Java compiler.

Bridge Methods

When compiling a class or interface that extends a parameterized class or implements a parameterized interface, the compiler may need to create a synthetic method, called a *bridge method*, as part of the type erasure process. You normally don't need to worry about bridge methods, but you might be puzzled if one appears in a stack trace.

After type erasure, the `Node` and `MyNode` classes become:

```
public class Node {

    private Object data;

    public Node(Object data) { this.data = data; }

    public void setData(Object data) {
        System.out.println("Node.setData");
        this.data = data;
    }
}

public class MyNode extends Node {

    public MyNode(Integer data) { super(data); }

    public void setData(Integer data) {
        System.out.println(Integer data);
        super.setData(data);
    }
}
```

After type erasure, the method signatures do not not match. The `Node` method becomes `setData(Object)` and the `MyNode` method becomes `setData(Integer)`. Therefore, the `MyNode` `setData` method does not override the `Node` `setData` method.

To solve this problem and preserve the polymorphism of generic types after type erasure, a Java compiler generates a bridge method to ensure that subtyping works as expected. For the `MyNode` class, the compiler generates the following bridge method for `setData`:

```
class MyNode extends Node {

    // Bridge method generated by the compiler
    //
    public void setData(Object data) {
        setData((Integer) data);
    }

    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }

    // ...
}
```

As you can see, the bridge method, which has the same method signature as the `Node` class's `setData` method after type erasure, delegates to the original `setData` method.

Non-Reifiable Types

The section Type Erasure discusses the process where the compiler removes information related to type parameters and type arguments. Type erasure has consequences related to variable arguments (also known as *varargs*) methods whose varargs formal parameter has a non-reifiable type. See the section Arbitrary Number of Arguments in Passing Information to a Method or a Constructor for more information about varargs methods.

This page covers the following topics:

- Non-Reifiable Types
- Heap Pollution
- Potential Vulnerabilities of Varargs Methods with Non-Reifiable Formal Parameters
- Preventing Warnings from Varargs Methods with Non-Reifiable Formal Parameters

Non-Reifiable Types

A *reifiable* type is a type whose type information is fully available at runtime. This includes primitives, non-generic types, raw types, and invocations of unbound wildcards.

Non-reifiable types are types where information has been removed at compile-time by type erasure invocations of generic types that are not defined as unbounded wildcards. A non-reifiable type does not have all of its information available at runtime. Examples of non-reifiable types are `List<String>` and `List<Number>`; the JVM cannot tell the difference between these types at runtime. As shown in Restrictions on Generics, there are certain situations where non-reifiable types cannot be used: in an `instanceof` expression, for example, or as an element in an array.

Heap Pollution

Heap pollution occurs when a variable of a parameterized type refers to an object that is not of that parameterized type. This situation occurs if the program performed some operation that gives rise to an unchecked warning at compile-time. An *unchecked warning* is generated if, either at compile-time (within the limits of the compile-time type checking rules) or at runtime, the correctness of an operation involving a parameterized type (for example, a cast or method call) cannot be verified. For example, heap pollution occurs when mixing raw types and parameterized types, or when performing unchecked casts.

In normal situations, when all code is compiled at the same time, the compiler issues an unchecked warning to draw your attention to potential heap pollution. If you compile sections of your code separately, it is difficult to detect the potential risk of heap pollution. If you ensure that your code compiles without warnings, then no heap pollution can occur.

Potential Vulnerabilities of Varargs Methods with Non-Reifiable Formal Parameters

Generic methods that include vararg input parameters can cause heap pollution.

Consider the following `ArrayBuilder` class:

```
public class ArrayBuilder {

  public static <T> void addToList (List<T> listArg, T... elements)
{
    for (T x : elements) {
      listArg.add(x);
    }
  }

  public static void faultyMethod(List<String>... l) {
    Object[] objectArray = l;     // Valid
    objectArray[0] = Arrays.asList(42);
    String s = l[0].get(0);       // ClassCastException thrown here
  }

}
```

The following example, `HeapPollutionExample` uses the `ArrayBuiler` class:

```
public class HeapPollutionExample {

  public static void main(String[] args) {

    List<String> stringListA = new ArrayList<String>();
    List<String> stringListB = new ArrayList<String>();

    ArrayBuilder.addToList(stringListA, "Seven", "Eight", "Nine");
    ArrayBuilder.addToList(stringListA, "Ten", "Eleven", "Twelve");
    List<List<String>> listOfStringLists =
      new ArrayList<List<String>>();
    ArrayBuilder.addToList(listOfStringLists,
      stringListA, stringListB);

    ArrayBuilder.faultyMethod(Arrays.asList("Hello!"), Arrays.asList("World!"));
  }
}
```

When compiled, the following warning is produced by the definition of the `ArrayBuilder.addToList` method:

```
warning: [varargs] Possible heap pollution from parameterized vararg type T
```

When the compiler encounters a varargs method, it translates the varargs formal parameter into an array. However, the Java programming language does not permit the creation of arrays of parameterized types. In the method `ArrayBuilder.addToList`, the compiler translates the varargs formal parameter `T... elements` to the formal parameter `T[] elements`, an array. However, because of type erasure, the compiler converts the varargs formal parameter to `Object[] elements`. Consequently, there is a possibility of heap pollution.

The following statement assigns the varargs formal parameter `l` to the `Object` array `objectArgs`:

```
Object[] objectArray = l;
```

This statement can potentially introduce heap pollution. A value that does match the parameterized type of the varargs formal parameter `l` can be assigned to the variable `objectArray`, and thus can be assigned to `l`. However, the compiler does not generate an unchecked warning at this statement. The compiler has already generated a warning when it translated the varargs formal parameter `List<String>... l` to the formal parameter `List[] l`. This statement is valid; the variable `l` has the type `List[]`, which is a subtype of `Object[]`.

Consequently, the compiler does not issue a warning or error if you assign a `List` object of any type to any array component of the `objectArray` array as shown by this statement:

```
objectArray[0] = Arrays.asList(42);
```

This statement assigns to the first array component of the `objectArray` array with a `List` object that contains one object of type `Integer`.

Suppose you invoke `ArrayBuilder.faultyMethod` with the following statement:

```
ArrayBuilder.faultyMethod(Arrays.asList("Hello!"), Arrays.asList("World!"));
```

At runtime, the JVM throws a `ClassCastException` at the following statement:

```
// ClassCastException thrown here
String s = l[0].get(0);
```

The object stored in the first array component of the variable `l` has the type `List<Integer>`, but this statement is expecting an object of type `List<String>`.

Prevent Warnings from Varargs Methods with Non-Reifiable Formal Parameters

If you declare a varargs method that has parameters of a parameterized type, and you ensure that the body of the method does not throw a `ClassCastException` or other similar exception due to improper handling of the varargs formal parameter, you can prevent the warning that the compiler generates for these kinds of varargs methods by adding the following annotation to static and non-constructor method declarations:

```
@SafeVarargs
```

The `@SafeVarargs` annotation is a documented part of the method's contract; this annotation asserts that the implementation of the method will not improperly handle the varargs formal parameter.

It is also possible, though less desirable, to suppress such warnings by adding the following to the method declaration:

```
@SuppressWarnings({"unchecked", "varargs"})
```

However, this approach does not suppress warnings generated from the method's call site. If you are unfamiliar with the `@SuppressWarnings` syntax, see Annotations.

Restrictions on Generics

To use Java generics effectively, you must consider the following restrictions:

- Cannot Instantiate Generic Types with Primitive Types
- Cannot Create Instances of Type Parameters
- Cannot Declare Static Fields Whose Types are Type Parameters
- Cannot Use Casts or `instanceof` With Parameterized Types
- Cannot Create Arrays of Parameterized Types
- Cannot Create, Catch, or Throw Objects of Parameterized Types
- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

Cannot Instantiate Generic Types with Primitive Types

Consider the following parameterized type:

```
class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    // ...
}
```

When creating a `Pair` object, you cannot substitute a primitive type for the type parameter `K` or `V`:

```
Pair<int, char> p = new Pair<>(8, 'a');  // compile-time error
```

You can substitute only non-primitive types for the type parameters `K` and `V`:

```
Pair<Integer, Character> p = new Pair<>(8, 'a');
```

Note that the Java compiler autoboxes `8` to `Integer.valueOf(8)` and 'a' to `Character('a')`:

```
Pair<Integer, Character> p = new Pair<>(Integer.valueOf(8), new Character('a'));
```

For more information on autoboxing, see Autoboxing and Unboxing in the Numbers and Strings lesson.

Cannot Create Instances of Type Parameters

You cannot create an instance of a type parameter. For example, the following code causes a compile-time error:

```
public static <E> void append(List<E> list) {
    E elem = new E();  // compile-time error
    list.add(elem);
}
```

As a workaround, you can create an object of a type parameter through reflection:

```
public static <E> void append(List<E> list, Class<E> cls) throws Exception {
    E elem = cls.newInstance();   // OK
    list.add(elem);
}
```

You can invoke the `append` method as follows:

```
List<String> ls = new ArrayList<>();
append(ls, String.class);
```

Cannot Declare Static Fields Whose Types are Type Parameters

A class's static field is a class-level variable shared by all non-static objects of the class. Hence, static fields of type parameters are not allowed. Consider the following class:

```
public class MobileDevice<T> {
    private static T os;

    // ...
}
```

If static fields of type parameters were allowed, then the following code would be confused:

```
MobileDevice<Smartphone> phone = new MobileDevice<>();
MobileDevice<Pager> pager = new MobileDevice<>();
MobileDevice<TabletPC> pc = new MobileDevice<>();
```

Because the static field `os` is shared by `phone`, `pager`, and `pc`, what is the actual type of `os`? It cannot be `Smartphone`, `Pager`, and `TabletPC` at the same time. You cannot, therefore, create static fields of type parameters.

Cannot Use Casts or `instanceof` with Parameterized Types

Because the Java compiler erases all type parameters in generic code, you cannot verify which parameterized type for a generic type is being used at runtime:

```
public static <E> void rtti(List<E> list) {
    if (list instanceof ArrayList<Integer>) {  // compile-time error
        // ...
    }
}
```

The set of parameterized types passed to the `rtti` method is:

```
S = { ArrayList<Integer>, ArrayList<String> LinkedList<Character>, ... }
```

The runtime does not keep track of type parameters, so it cannot tell the difference between an `ArrayList<Integer>` and an `ArrayList<String>`. The most you can do is to use an unbounded wildcard to verify that the list is an `ArrayList`:

```
public static void rtti(List<?> list) {
    if (list instanceof ArrayList<?>) {  // OK; instanceof requires a reifiable type
        // ...
    }
}
```

Typically, you cannot cast to a parameterized type unless it is parameterized by unbounded wildcards. For example:

```
List<Integer> li = new ArrayList<>();
List<Number>  ln = (List<Number>) li;  // compile-time error
```

However, in some cases the compiler knows that a type parameter is always valid and allows the cast. For example:

```
List<String> l1 = ...;
ArrayList<String> l2 = (ArrayList<String>)l1;  // OK
```

Cannot Create Arrays of Parameterized Types

You cannot create arrays of parameterized types. For example, the following code does not compile:

```
List<Integer>[] arrayOfLists = new List<Integer>[2];  // compile-time error
```

The following code illustrates what happens when different types are inserted into an array:

```
Object[] strings = new String[2];
strings[0] = "hi";   // OK
strings[1] = 100;    // An ArrayStoreException is thrown.
```

If you try the same thing with a generic list, there would be a problem:

```
Object[] stringLists = new List<String>[];  // compiler error, but pretend it's allowed
```

```
stringLists[0] = new ArrayList<String>();   // OK
stringLists[1] = new ArrayList<Integer>();  // An ArrayStoreException should be thrown,
                                            // but the runtime can't detect it.
```

If arrays of parameterized lists were allowed, the previous code would fail to throw the desired `ArrayStoreException`.

Cannot Create, Catch, or Throw Objects of Parameterized Types

A generic class cannot extend the `Throwable` class directly or indirectly. For example, the following classes will not compile:

```
// Extends Throwable indirectly
class MathException<T> extends Exception { /* ... */ }    // compile-time error

// Extends Throwable directly
class QueueFullException<T> extends Throwable { /* ... */ // compile-time error
```

A method cannot catch an instance of a type parameter:

```
public static <T extends Exception, J> void execute(List<J> jobs) {
    try {
        for (J job : jobs)
            // ...
    } catch (T e) {   // compile-time error
        // ...
    }
}
```

You can, however, use a type parameter in a `throws` clause:

```
class Parser<T extends Exception> {
    public void parse(File file) throws T {     // OK
        // ...
    }
}
```

Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

A class cannot have two overloaded methods that will have the same signature after type erasure.

```
public class Example {
    public void print(Set<String> strSet) { }
    public void print(Set<Integer> intSet) { }
}
```

The overloads would all share the same classfile representation and will generate a compile-time error.

Questions and Exercises: Generics

1. Write a generic method to count the number of elements in a collection that have a specific property (for example, odd integers, prime numbers, palindromes).

2. Will the following class compile? If not, why?

```
public final class Algorithm {
    public static T max(T x, T y) {
        return x > y ? x : y;
    }
}
```

3. Write a generic method to exchange the positions of two different elements in an array.

4. If the compiler erases all type parameters at compile time, why should you use generics?

5. What is the following class converted to after type erasure?

```
public class Pair<K, V> {

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey(); { return key; }
    public V getValue(); { return value; }

    public void setKey(K key)     { this.key = key; }
    public void setValue(V value) { this.value = value; }

    private K key;
    private V value;
}
```

6. What is the following method converted to after type erasure?

```
public static <T extends Comparable<T>>
    int findFirstGreaterThan(T[] at, T elem) {
    // ...
}
```

7. Will the following method compile? If not, why?

```
public static void print(List<? extends Number> list) {
    for (Number n : list)
        System.out.print(n + " ");
    System.out.println();
}
```

8. Write a generic method to find the maximal element in the range `[begin, end)` of a list.

9. Will the following class compile? If not, why?

```
public class Singleton<T> {

    public static T getInstance() {
        if (instance == null)
            instance = new Singleton<T>();
        return instance;
    }

    private static T instance = null;
}
```

10. Given the following classes:

```
class Shape { /* ... */ }
class Circle extends Shape { /* ... */ }
class Rectangle extends Shape { /* ... */ }

class Node<T> { /* ... */ }
```

Will the following code compile? If not, why?

```
Node<Circle> nc = new Node<>();
Node<Shape>  ns = nc;
```

11. Consider this class:

```
class Node<T> implements Comparable<T> {
    public int compareTo(T obj) { /* ... */ }
    // ...
}
```

Will the following code compile? If not, why?

```
Node<String> node = new Node<>();
Comparable<String> comp = node;
```

12. How do you invoke the following method to find the first integer in a list that is relatively prime to a list of specified integers?

```
public static <T>
    int findFirst(List<T> list, int begin, int end, UnaryPredicate<T> p)
```

Note that two integers *a* and *b* are relatively prime if gcd(*a, b*) = 1, where gcd is short for greatest common divisor.

Check your answers.

1. Write a generic method to count the number of elements in a collection that have a specific property (for example, odd integers, prime numbers, palindromes).

Answer:

```
public final class Algorithm {
    public static <T> int countIf(Collection<T> c, UnaryPredicate<T> p) {

        int count = 0;
        for (T elem : c)
            if (p.test(elem))
                ++count;
        return count;
    }
}
```

where the generic `UnaryPredicate` interface is defined as follows:

```
public interface UnaryPredicate<T> {
    public boolean test(T obj);
}
```

For example, the following program counts the number of odd integers in an integer list:

```
import java.util.*;

class OddPredicate implements UnaryPredicate<Integer> {
    public boolean test(Integer i) { return i % 2 != 0; }
}

public class Test {
    public static void main(String[] args) {
        Collection<Integer> ci = Arrays.asList(1, 2, 3, 4);
        int count = Algorithm.countIf(ci, new OddPredicate());
        System.out.println("Number of odd integers = " + count);
    }
}
```

The program prints:

```
Number of odd integers = 2
```

2. Will the following class compile? If not, why?

```
public final class Algorithm {
    public static T max(T x, T y) {
        return x > y ? x : y;
    }
}
```

Answer: No. The greater than (>) operator applies only to primitive numeric types.

3. Write a generic method to exchange the positions of two different elements in an array.

Answer:

```
public final class Algorithm {
    public static <T> void swap(T[] a, int i, int j) {
        T temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}
```

4. If the compiler erases all type parameters at compile time, why should you use generics?

Answer: You should use generics because:

- The Java compiler enforces tighter type checks on generic code at compile time.
- Generics support programming types as parameters.
- Generics enable you to implement generic algorithms.

5. What is the following class converted to after type erasure?

```
public class Pair<K, V> {

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey(); { return key; }
    public V getValue(); { return value; }

    public void setKey(K key)     { this.key = key; }
    public void setValue(V value) { this.value = value; }

    private K key;
    private V value;
}
```

Answer:

```
public class Pair {

    public Pair(Object key, Object value) {
        this.key = key;
        this.value = value;
    }

    public Object getKey()   { return key; }
    public Object getValue() { return value; }

    public void setKey(Object key)     { this.key = key; }
    public void setValue(Object value) { this.value = value; }

    private Object key;
    private Object value;
}
```

6. What is the following method converted to after type erasure?

```
public static <T extends Comparable<T>>
    int findFirstGreaterThan(T[] at, T elem) {
    // ...
}
```

Answer:

```
public static int findFirstGreaterThan(Comparable[] at, Comparable elem) {
    // ...
    }
```

7. Will the following method compile? If not, why?

```
public static void print(List<? extends Number> list) {
    for (Number n : list)
        System.out.print(n + " ");
    System.out.println();
}
```

Answer: Yes.

8. Write a generic method to find the maximal element in the range `[begin, end)` of a list.

Answer:

```
import java.util.*;

public final class Algorithm {
    public static <T extends Object & Comparable<? super T>>
        T max(List<? extends T> list, int begin, int end) {

        T maxElem = list.get(begin);

        for (++begin; begin < end; ++begin)
            if (maxElem.compareTo(list.get(begin)) < 0)
                maxElem = list.get(begin);
        return maxElem;
    }
}
```

9. Will the following class compile? If not, why?

```
public class Singleton<T> {

    public static T getInstance() {
        if (instance == null)
            instance = new Singleton<T>();

        return instance;
    }

    private static T instance = null;
}
```

Answer: No. You cannot create a static field of the type parameter `T`.

10. Given the following classes:

```
class Shape { /* ... */ }
class Circle extends Shape { /* ... */ }
class Rectangle extends Shape { /* ... */ }

class Node<T> { /* ... */ }
```

Will the following code compile? If not, why?

```
Node<Circle> nc = new Node<>();
Node<Shape>  ns = nc;
```

Answer: No. Because `Node<Circle>` is not a subtype of `Node<Shape>`.

11. Consider this class:

```
class Node<T> implements Comparable<T> {
    public int compareTo(T obj) { /* ... */ }
    // ...
}
```

Will the following code compile? If not, why?

Answer: Yes.

```
Node<String> node = new Node<>();
Comparable<String> comp = node;
```

12. How do you invoke the following method to find the first integer in a list that is relatively prime to a list of specified integers?

```
public static <T>
    int findFirst(List<T> list, int begin, int end, UnaryPredicate<T> p)
```

Note that two integers *a* and *b* are relatively prime if gcd(*a, b*) = 1, where gcd is short for greatest common divisor.

Answer:

```
import java.util.*;

public final class Algorithm {

    public static <T>
        int findFirst(List<T> list, int begin, int end, UnaryPredicate<T> p) {

        for (; begin < end; ++begin)
            if (p.test(list.get(begin)))
                return begin;
        return -1;
    }

    // x > 0 and y > 0
    public static int gcd(int x, int y) {
        for (int r; (r = x % y) != 0; x = y, y = r) { }
            return y;
    }
}
```

The generic `UnaryPredicate` interface is defined as follows:

```
public interface UnaryPredicate<T> {
    public boolean test(T obj);
}
```

The following program tests the `findFirst` method:

```
import java.util.*;

class RelativelyPrimePredicate implements UnaryPredicate<Integer> {
    public RelativelyPrimePredicate(Collection<Integer> c) {
        this.c = c;
    }

    public boolean test(Integer x) {
        for (Integer i : c)
            if (Algorithm.gcd(x, i) != 1)
                return false;

        return c.size() > 0;
    }

    private Collection<Integer> c;
}

public class Test {
    public static void main(String[] args) throws Exception {

        List<Integer> li = Arrays.asList(3, 4, 6, 8, 11, 15, 28, 32);
        Collection<Integer> c = Arrays.asList(7, 18, 19, 25);
        UnaryPredicate<Integer> p = new RelativelyPrimePredicate(c);

        int i = ALgorithm.findFirst(li, 0, li.size(), p);

        if (i != -1) {
            System.out.print(li.get(i) + " is relatively prime to ");
            for (Integer k : c)
                System.out.print(k + " ");
            System.out.println();
        }
    }
}
```

The program prints:

```
11 is relatively prime to 7 18 19 25
```

Lesson: Packages

This lesson explains how to bundle classes and interfaces into packages, how to use classes that are in packages, and how to arrange your file system so that the compiler can find your source files.

Note: See online version of topics in this ebook to download complete source code.

Creating and Using Packages

To make types easier to find and use, to avoid naming conflicts, and to control access, programmers bundle groups of related types into packages.

Definition: A *package* is a grouping of related types providing access protection and name space management. Note that *types* refers to classes, interfaces, enumerations, and annotation types. Enumerations and annotation types are special kinds of classes and interfaces, respectively, so *types* are often referred to in this lesson simply as *classes and interfaces*.

The types that are part of the Java platform are members of various packages that bundle classes by function: fundamental classes are in `java.lang`, classes for reading and writing (input and output) are in `java.io`, and so on. You can put your types in packages too.

Suppose you write a group of classes that represent graphic objects, such as circles, rectangles, lines, and points. You also write an interface, `Draggable`, that classes implement if they can be dragged with the mouse.

```
//in the Draggable.java file
public interface Draggable {
    ...
}

//in the Graphic.java file
public abstract class Graphic {
    ...
}

//in the Circle.java file
public class Circle extends Graphic
    implements Draggable {
    . . .
}

//in the Rectangle.java file
public class Rectangle extends Graphic
    implements Draggable {
    . . .
}

//in the Point.java file
public class Point extends Graphic
    implements Draggable {
    . . .
}

//in the Line.java file
public class Line extends Graphic
    implements Draggable {
    . . .
}
```

You should bundle these classes and the interface in a package for several reasons, including the following:

- You and other programmers can easily determine that these types are related.
- You and other programmers know where to find types that can provide graphics-related functions.
- The names of your types won't conflict with the type names in other packages because the package creates a new namespace.
- You can allow types within the package to have unrestricted access to one another yet still restrict access for types outside the package.

Creating a Package

To create a package, you choose a name for the package (naming conventions are discussed in the next section) and put a `package` statement with that name at the top of *every source file* that contains the types (classes, interfaces, enumerations, and annotation types) that you want to include in the package.

The package statement (for example, `package graphics;`) must be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

Note: If you put multiple types in a single source file, only one can be `public`, and it must have the same name as the source file. For example, you can define `public class Circle` in the file `Circle.java`, define `public interface Draggable` in the file `Draggable.java`, define `public enum Day` in the file `Day.java`, and so forth.

You can include non-public types in the same file as a public type (this is strongly discouraged, unless the non-public types are small and closely related to the public type), but only the public type will be accessible from outside of the package. All the top-level, non-public types will be *package private*.

If you put the graphics interface and classes listed in the preceding section in a package called `graphics`, you would need six source files, like this:

```
//in the Draggable.java file
package graphics;
public interface Draggable {
    . . .
}

//in the Graphic.java file
package graphics;
public abstract class Graphic {
    . . .
}

//in the Circle.java file
package graphics;
public class Circle extends Graphic
    implements Draggable {
    . . .
}

//in the Rectangle.java file
package graphics;
public class Rectangle extends Graphic
    implements Draggable {
    . . .
}

//in the Point.java file
package graphics;
public class Point extends Graphic
    implements Draggable {
    . . .
}

//in the Line.java file
```

```
package graphics;
public class Line extends Graphic
    implements Draggable {
    . . .
}
```

If you do not use a `package` statement, your type ends up in an unnamed package. Generally speaking, an unnamed package is only for small or temporary applications or when you are just beginning the development process. Otherwise, classes and interfaces belong in named packages.

Naming a Package

With programmers worldwide writing classes and interfaces using the Java programming language, it is likely that many programmers will use the same name for different types. In fact, the previous example does just that: It defines a `Rectangle` class when there is already a `Rectangle` class in the `java.awt` package. Still, the compiler allows both classes to have the same name if they are in different packages. The fully qualified name of each `Rectangle` class includes the package name. That is, the fully qualified name of the `Rectangle` class in the `graphics` package is `graphics.Rectangle`, and the fully qualified name of the `Rectangle` class in the `java.awt` package is `java.awt.Rectangle`.

This works well unless two independent programmers use the same name for their packages. What prevents this problem? Convention.

Naming Conventions

Package names are written in all lower case to avoid conflict with the names of classes or interfaces.

Companies use their reversed Internet domain name to begin their package names—for example, `com.example.mypackage` for a package named `mypackage` created by a programmer at `example.com`.

Name collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region or the project name after the company name (for example, `com.example.region.mypackage`).

Packages in the Java language itself begin with `java.` or `javax.`

In some cases, the internet domain name may not be a valid package name. This can occur if the domain name contains a hyphen or other special character, if the package name begins with a digit or other character that is illegal to use as the beginning of a Java name, or if the package name contains a reserved Java keyword, such as "int". In this event, the suggested convention is to add an underscore. For example:

Legalizing Package Names

Domain Name	Package Name Prefix
`hyphenated-name.example.org`	`org.example.hyphenated_name`
`example.int`	`int_.example`
`123name.example.com`	`com.example._123name`

Using Package Members

The types that comprise a package are known as the *package members*.

To use a `public` package member from outside its package, you must do one of the following:

- Refer to the member by its fully qualified name
- Import the package member
- Import the member's entire package

Each is appropriate for different situations, as explained in the sections that follow.

Referring to a Package Member by Its Qualified Name

So far, most of the examples in this tutorial have referred to types by their simple names, such as `Rectangle` and `StackOfInts`. You can use a package member's simple name if the code you are writing is in the same package as that member or if that member has been imported.

However, if you are trying to use a member from a different package and that package has not been imported, you must use the member's fully qualified name, which includes the package name. Here is the fully qualified name for the `Rectangle` class declared in the `graphics` package in the previous example.

```
graphics.Rectangle
```

You could use this qualified name to create an instance of `graphics.Rectangle`:

```
graphics.Rectangle myRect = new graphics.Rectangle();
```

Qualified names are all right for infrequent use. When a name is used repetitively, however, typing the name repeatedly becomes tedious and the code becomes difficult to read. As an alternative, you can *import* the member or its package and then use its simple name.

Importing a Package Member

To import a specific member into the current file, put an `import` statement at the beginning of the file before any type definitions but after the `package` statement, if there is one. Here's how you would import the `Rectangle` class from the `graphics` package created in the previous section.

```
import graphics.Rectangle;
```

Now you can refer to the `Rectangle` class by its simple name.

```
Rectangle myRectangle = new Rectangle();
```

This approach works well if you use just a few members from the `graphics` package. But if you use many types from a package, you should import the entire package.

Importing an Entire Package

To import all the types contained in a particular package, use the `import` statement with the asterisk `(*)` wildcard character.

```
import graphics.*;
```

Now you can refer to any class or interface in the `graphics` package by its simple name.

```
Circle myCircle = new Circle();
Rectangle myRectangle = new Rectangle();
```

The asterisk in the `import` statement can be used only to specify all the classes within a package, as shown here. It cannot be used to match a subset of the classes in a package. For example, the following does not match all the classes in the `graphics` package that begin with `A`.

```
// does not work
import graphics.A*;
```

Instead, it generates a compiler error. With the `import` statement, you generally import only a single package member or an entire package.

Note: Another, less common form of `import` allows you to import the public nested classes of an enclosing class. For example, if the `graphics.Rectangle` class contained useful nested classes, such as `Rectangle.DoubleWide` and `Rectangle.Square`, you could import `Rectangle` and its nested classes by using the following *two* statements.

```
import graphics.Rectangle;
import graphics.Rectangle.*;
```

Be aware that the second import statement will *not* import `Rectangle`.

Another less common form of `import`, the *static import statement*, will be discussed at the end of this section.

For convenience, the Java compiler automatically imports two entire packages for each source file: (1) the `java.lang` package and (2) the current package (the package for the current file).

Apparent Hierarchies of Packages

At first, packages appear to be hierarchical, but they are not. For example, the Java API includes a java.awt package, a java.awt.color package, a java.awt.font package, and many others that begin with java.awt. However, the java.awt.color package, the java.awt.font package, and other java.awt.xxxx packages are *not included* in the java.awt package. The prefix java.awt (the Java Abstract Window Toolkit) is used for a number of related packages to make the relationship evident, but not to show inclusion.

Importing java.awt.* imports all of the types in the java.awt package, but it *does not import* java.awt.color, java.awt.font, or any other java.awt.xxxx packages. If you plan to use the classes and other types in java.awt.color as well as those in java.awt, you must import both packages with all their files:

```
import java.awt.*;
import java.awt.color.*;
```

Name Ambiguities

If a member in one package shares its name with a member in another package and both packages are imported, you must refer to each member by its qualified name. For example, the graphics package defined a class named Rectangle. The java.awt package also contains a Rectangle class. If both graphics and java.awt have been imported, the following is ambiguous.

```
Rectangle rect;
```

In such a situation, you have to use the member's fully qualified name to indicate exactly which Rectangle class you want. For example,

```
graphics.Rectangle rect;
```

The Static Import Statement

There are situations where you need frequent access to static final fields (constants) and static methods from one or two classes. Prefixing the name of these classes over and over can result in cluttered code. The *static import* statement gives you a way to import the constants and static methods that you want to use so that you do not need to prefix the name of their class.

The java.lang.Math class defines the PI constant and many static methods, including methods for calculating sines, cosines, tangents, square roots, maxima, minima, exponents, and many more. For example,

```
public static final double PI
    = 3.141592653589793;
public static double cos(double a)
{
    ...
}
```

Ordinarily, to use these objects from another class, you prefix the class name, as follows.

```
double r = Math.cos(Math.PI * theta);
```

You can use the static import statement to import the static members of java.lang.Math so that you don't need to prefix the class name, `Math`. The static members of `Math` can be imported either individually:

```
import static java.lang.Math.PI;
```

or as a group:

```
import static java.lang.Math.*;
```

Once they have been imported, the static members can be used without qualification. For example, the previous code snippet would become:

```
double r = cos(PI * theta);
```

Obviously, you can write your own classes that contain constants and static methods that you use frequently, and then use the static import statement. For example,

```
import static mypackage.MyConstants.*;
```

Note: Use static import very sparingly. Overusing static import can result in code that is difficult to read and maintain, because readers of the code won't know which class defines a particular static object. Used properly, static import makes code more readable by removing class name repetition.

Managing Source and Class Files

Many implementations of the Java platform rely on hierarchical file systems to manage source and class files, although *The Java Language Specification* does not require this. The strategy is as follows.

Put the source code for a class, interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is `.java`. For example:

```
//in the Rectangle.java file
package graphics;
public class Rectangle {
   ...
}
```

Then, put the source file in a directory whose name reflects the name of the package to which the type belongs:

```
.....\graphics\Rectangle.java
```

The qualified name of the package member and the path name to the file are parallel, assuming the Microsoft Windows file name separator backslash (for UNIX, use the forward slash).

- **class name** `graphics.Rectangle`
- **pathname to file** `graphics\Rectangle.java`

As you should recall, by convention a company uses its reversed Internet domain name for its package names. The Example company, whose Internet domain name is `example.com`, would precede all its package names with `com.example`. Each component of the package name corresponds to a subdirectory. So, if the Example company had a `com.example.graphics` package that contained a `Rectangle.java` source file, it would be contained in a series of subdirectories like this:

```
....\com\example\graphics\Rectangle.java
```

When you compile a source file, the compiler creates a different output file for each type defined in it. The base name of the output file is the name of the type, and its extension is `.class`. For example, if the source file is like this

```
//in the Rectangle.java file
package com.example.graphics;
public class Rectangle {
      . . .
}

class Helper{
      . . .
}
```

then the compiled files will be located at:

```
<path to the parent directory of the output files>\com\example\graphics\Rectangle.class
<path to the parent directory of the output files>\com\example\graphics\Helper.class
```

Like the `.java` source files, the compiled `.class` files should be in a series of directories that reflect the package name. However, the path to the `.class` files does not have to be the same as the path to the `.java` source files. You can arrange your source and class directories separately, as:

```
<path_one>\sources\com\example\graphics\Rectangle.java

<path_two>\classes\com\example\graphics\Rectangle.class
```

By doing this, you can give the `classes` directory to other programmers without revealing your sources. You also need to manage source and class files in this manner so that the compiler and the Java Virtual Machine (JVM) can find all the types your program uses.

The full path to the `classes` directory, `<path_two>\classes`, is called the *class path*, and is set with the `CLASSPATH` system variable. Both the compiler and the JVM construct the path to your `.class` files by adding the package name to the class path. For example, if

```
<path_two>\classes
```

is your class path, and the package name is

```
com.example.graphics,
```

then the compiler and JVM look for `.class files` in

```
<path_two>\classes\com\example\graphics.
```

A class path may include several paths, separated by a semicolon (Windows) or colon (UNIX). By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in your class path.

Setting the CLASSPATH System Variable

To display the current `CLASSPATH` variable, use these commands in Windows and UNIX (Bourne shell):

```
In Windows:C:\> set CLASSPATH
In UNIX:% echo $CLASSPATH
```

To delete the current contents of the `CLASSPATH` variable, use these commands:

```
In Windows:C:\> set CLASSPATH=
In UNIX:% unset CLASSPATH; export CLASSPATH
```

To set the `CLASSPATH` variable, use these commands (for example):

```
In Windows:C:\> set CLASSPATH=C:\users\george\java\classes
In UNIX:% CLASSPATH=/home/george/java/classes; export CLASSPATH
```

Summary of Creating and Using Packages

To create a package for a type, put a `package` statement as the first statement in the source file that contains the type (class, interface, enumeration, or annotation type).

To use a public type that's in a different package, you have three choices: (1) use the fully qualified name of the type, (2) import the type, or (3) import the entire package of which the type is a member.

The path names for a package's source and class files mirror the name of the package.

You might have to set your `CLASSPATH` so that the compiler and the JVM can find the `.class` files for your types.

Questions and Exercises: Creating and Using Packages

Questions

Assume you have written some classes. Belatedly, you decide they should be split into three packages, as listed in the following table. Furthermore, assume the classes are currently in the default package (they have no `package` statements).

Destination Packages

Package Name	**Class Name**
`mygame.server`	`Server`
`mygame.shared`	`Utilities`
`mygame.client`	`Client`

1. Which line of code will you need to add to each source file to put each class in the right package?
2. To adhere to the directory structure, you will need to create some subdirectories in the development directory and put source files in the correct subdirectories. What subdirectories must you create? Which subdirectory does each source file go in?
3. Do you think you'll need to make any other changes to the source files to make them compile correctly? If so, what?

Exercises

Download the source files as listed here.

- `Client`
- `Server`
- `Utilities`

1. Implement the changes you proposed in questions 1 through 3 using the source files you just downloaded.
2. Compile the revised source files. (*Hint:* If you're invoking the compiler from the command line (as opposed to using a builder), invoke the compiler from the directory that contains the `mygame` directory you just created.)

Check your answers.

Answers

Question 1: Assume that you have written some classes. Belatedly, you decide that they should be split into three packages, as listed in the table below. Furthermore, assume that the classes are currently in the default package (they have no `package` statements).

Package Name	Class Name
`mygame.server`	`Server`
`mygame.shared`	`Utilities`
`mygame.client`	`Client`

a. What line of code will you need to add to each source file to put each class in the right package?
Answer 1a: The first line of each file must specify the package:

In `Client.java` add:
```
    package mygame.client;
```
In `Server.java` add:
```
    package mygame.server;:
```
In `Utilities.java` add:
```
    package mygame.shared;
```

b. To adhere to the directory structure, you will need to create some subdirectories in your development directory, and put source files in the correct subdirectories. What subdirectories must you create? Which subdirectory does each source file go in?
Answer 1b: Within the `mygame` directory, you need to create three subdirectories: `client`, `server`, and `shared`.

In `mygame/client/` place:
```
    Client.java
```
In `mygame/server/` place:
```
    Server.java
```
In `mygame/shared/` place:
```
    Utilities.java
```

c. Do you think you'll need to make any other changes to the source files to make them compile correctly? If so, what?
Answer 1c: Yes, you need to add import statements. `Client.java` and `Server.java` need to import the `Utilities` class, which they can do in one of two ways:

```
import mygame.shared.*;
       --or--
import mygame.shared.Utilities;
```

Also, `Server.java` needs to import the `Client` class:

```
import mygame.client.Client;
```

Exercises

Exercise 1: Download three source files:

- `Client`
- `Server`
- `Utilities`

a. Implement the changes you proposed in question 1, using the source files you just downloaded.
b. Compile the revised source files. (*Hint:* If you're invoking the compiler from the command line (as opposed to using a builder), invoke the compiler from the directory that contains the `mygame` directory you just created.) Answer 1: Download this zip file with the solution: `mygame.zip`
You might need to change your proposed import code to reflect our implementation.

Learning the Java Language: End of Trail

You have reached the end of the "Learning the Java Language" trail.

If you have comments or suggestions about this trail, use our feedback page to tell us about it.

Essential Classes: Learn about the most-used classes in the JDK APIs including `String`, `System`, `Thread` and the classes in `java.io`.

Creating a GUI With JFC/Swing: Once you know how to create applications or applets, follow this trail to learn how to create their user interfaces.

Custom Networking: If you're interested in writing applications or applets that use the network, follow this trail. You'll learn about URLs, sockets, datagrams, and security.