Contents

Chapter 1

Access Control

1.1 Introduction

Refactoring = re-writing working code to make it more readable, understandable and maintainable.
Refactoring happens and it requires to make sure it does not affect the client programmers which use a class or a library being re-factored.
Access Control provides a way to make sure the library creator can say what is available to the client programmers and what is not. The levels of Access Control from most access to least access are:

- public
- protected
- package access (which has no keyword)
- private

1.2 Package: The Library Unit

A package contains a group of classes, organized together under a single namespace.

A source-code file for java is commonly called *compilation unit*. Each compilation unit ends with the `.java` extension and inside the compilation unit there can be **only one public class** which **must have the same name of the compilation unit itself**.

If there are additional classes in the compilation unit, these classes are hidden from the world outside the package (because they are not public).

When a compilation unit is compiled, a `.class` file is generated for each class in the compilation unit.

A **library** is a group of these class files. If you want to say that all these components (each in its own separate `.java` and `.class` files) belong together, that is where the `package` keyword comes in.

If you use a package statement, it must appear as the first non-comment in the file. When you say:

```
package access;
```

you are stating that this compilation unit is part of a library named **access**. Put another way, you are saying that the public class name within this compilation unit is under the umbrella of the name **access**, and anyone who wants to use that name must either fully specify the name or use the **import** keyword in combination with **access**.

A package can be made up of many `.class` files; a logical thing to do is to place all the `.class` files for a particular package into a single directory. By convention, the first part of the package name is the reversed Internet domain name of the creator of the class. The second part of this trick is resolving the package name into a directory on your machine, so that when the Java program runs and it needs to load the `.class` file, it can locate the directory where the `.class` file resides.

The Java interpreter proceeds as follows. First, it finds the environment variable CLASSPATH (set via the operating system, and sometimes by the installation program that installs Java or a Java-based tool on your machine). CLASSPATH contains one or more directories that are used as roots in a search for `.class` files. Starting at that root, the interpreter will take the package name and replace each dot with a slash to generate a path name off of the CLASSPATH root (so package foo.bar.baz becomes foo\bar\baz).

Example:

CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT

(You can see that the CLASSPATH can contain a number of alternative search paths).

Folder where the `.class` files are saved is:

C:DOC\JavaT\net\mindview\simple
Package declaration is:

```
package net.mindview.simple;
```

1.3 Java Access Specifiers

The Java access specifiers **public**, **protected** and **private** are placed in front of each definition for each member in your class, whether it is a field or a method. Each access specifier only controls the access for that particular definition. If you don not provide an access specifier, it means "package access".

1.3.1 Package Access

The default access has no keyword but it is commonly referred as package access (and sometimes "friendly"). It means that all the other classes in the current package have access to that member, but to all the classes outside of this package, the member appears to be private. Since a compilation unit can belong only to a single package, all the classes within a single compilation unit are automatically available to each other via package access.

1.3.2 Public Access

When you use the **public** keyword, it means that the member declaration that immediately follows **public** is available to everyone, in particular to the client programmer who uses the library.

Notes:

- If **class** X in **package** P is **public**, to use it outside **package** P, the **import** statement to import class X is needed.
- Even if **class** X is public, it does not mean its methods or its members are by default public! Each method or member must be defined separately as public!

```
package access.dessert;

public class Cookie {
  public Cookie() {
   System.out.println("Cookie constructor");
  }
  void bite() { System.out.println("bite"); }
} ///:~


import access.dessert.*;

public class Dinner {
  public static void main(String[] args) {
    Cookie x = new Cookie();
```

```
    //! x.bite(); // Can't access
  }
} /* Output:
Cookie constructor
*///:~
```

1.3.3 Private Access

The **private** keyword means that no one can access that member except the class that contains that member inside methods of that class. Other classes in the same package cannot access **private** members.

1.3.4 Protected Access

Sometimes the creator of the base class would like to take a particular member and grant access to derived classes but not the world in general. That is what **protected** does. **protected** also gives package access, that's other classes in the same package may access protected elements.

1.4 Interface and Implementation

Access control is often referred as *implementation hiding.*
Wrapping data and methods within classes in combination with implementation hiding is often called *encapsulation.*
Access control puts boundaries within a data type for two important reasons.

- The first is to establish what the client programmers can and can not use.
- The second is to separate the interface from the implementation. If the structure is used in a set of programs, but client programmers can not do anything but send messages to the public interface, then you are free to change anything that is not public (e.g., package access, protected, or private) without breaking client code.

Chapter 2

Reusing Classes

2.1 Introduction: Composition and Inheritance

Java provides 2 main ways to reuse classes:

- The first is quite straightforward: you simply create objects of your existing class inside the new class. This is called *composition*, because the new class is composed of objects of existing classes. You are simply reusing the functionality of the code, not its form.
- The second approach creates a new class as a type of an existing class. You literally take the form of the existing class and add code to it without modifying the existing class. This technique is called *inheritance*, and the compiler does most of the work.

2.2 Composition Syntax

You simply place object references inside new classes.

```
package reusing;

//: reusing/SprinklerSystem.java
// Composition for code reuse.

class WaterSource {
  private String s;
  WaterSource() {
    System.out.println("WaterSource()");
    s = "Constructed";
  }
  public String toString() { return s; }
}
```

```
public class SprinklerSystem {
  private String valve1, valve2, valve3, valve4;
  private WaterSource source = new WaterSource();
  private int i;
  private float f;
  public String toString() {
    return
      "valve1 = " + valve1 + " " +
      "valve2 = " + valve2 + " " +
      "valve3 = " + valve3 + " " +
      "valve4 = " + valve4 + "\n" +
      "i = " + i + " " + "f = " + f + " " +
      "source = " + source;
  }
  public static void main(String[] args) {
    SprinklerSystem sprinklers = new SprinklerSystem();
    System.out.println(sprinklers);
  }
}
```

2.3 Inheritance Syntax

When you inherit, you say "This new class is like that old class."You state this in code before the opening brace of the class body, using the keyword **extends** followed by the name of the base class. When you do this, you automatically get all the fields and methods in the base class.

Both the base class and the derived class can contain a main( ) method. You can create a main( ) for each one of your classes; this technique of putting a main() in each class allows easy testing for each class. And you do not need to remove the main() when you are finished; you can leave it in for later testing.

The derived class automatically gets all the non private members of the base class, even if they are not explicitly defined in the derived class.

The derived class can take a method of the base class and modify it; but it is possible to use the keyword **super** to call the base-class version of a method.

When inheriting you are not restricted to using the methods of the base class. You can also add new methods to the derived class exactly the way you put any method in a class: Just define it.

```
import static net.mindview.util.Print.*;
```

```
class Cleanser {
  private String s = "Cleanser";
  public void append(String a) { s += a; }
  public void dilute() { append(" dilute()"); }
  public void apply() { append(" apply()"); }
  public void scrub() { append(" scrub()"); }
  public String toString() { return s; }
  public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    print(x);
  }
}

public class Detergent extends Cleanser {
  // Change a method:
  public void scrub() {
    append(" Detergent.scrub()");
    super.scrub(); // Call base-class version
  }
  // Add methods to the interface:
  public void foam() { append(" foam()"); }
  // Test the new class:
  public static void main(String[] args) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    print(x);
    print("Testing base class:");
    Cleanser.main(args);
  }
}
```

2.3.1 Inheritance and Initialization

When you create an object of the derived class, it contains within it a sub-object of the base class. This sub-object is the same as if you had created an object of the base class by itself.

Java automatically inserts calls to the base-class constructor in the derived-class constructor.

The construction happens from the base "outward" so the base class is ini-

tialized before the derived-class constructors can access it.

```
import static net.mindview.util.Print.*;

class Art {
  Art() { print("Art constructor"); }
}

class Drawing extends Art {
  Drawing() { print("Drawing constructor"); }
}

public class Cartoon extends Drawing {
  public Cartoon() { print("Cartoon constructor"); }
  public static void main(String[] args) {
    Cartoon x = new Cartoon();
  }
}
 /* Output:
Art constructor
Drawing constructor
Cartoon constructor
*///:
```

2.3.2 Combining Composition and Inheritance

It is very common to use composition and inheritance together.

2.4 Choosing Composition vs Inheritance

Composition is generally used when you want the features of an existing class inside your new class, but not its interface. That is, you embed an object so that you can use it to implement features in your new class, but the user of your new class sees the interface you have defined for the new class rather than the interface from the embedded object. For this effect, you embed private objects of existing classes inside your new class.

When you inherit, you take an existing class and make a special version of it. In general, this means that you are taking a general purpose class and specializing it for a particular need.

A practical way to choose is to think about the type of relationship between the classes; the *is-a* relationship is expressed with inheritance, and the *has-a* relationship is expressed with composition.

Another way to determine whether you should use composition or inheritance is to ask whether you will ever need to upcast from your new class to the base class. If you must upcast, then inheritance is necessary, but if you don not need to upcast, then you should look closely at whether you need inheritance.

2.5 Upcasting

The most important aspect of inheritance is not that it provides methods for the new class. It is the relationship expressed between the new class and the base class. This relationship can be summarized by saying, "The new class is a type of the existing class" and the language directly supports this relationship.

```
class Instrument {
  public void play() {}
  static void tune(Instrument i) {
    // ...
    i.play();
  }
}

// Wind objects are instruments
// because they have the same interface:
public class Wind extends Instrument {
  public static void main(String[] args) {
    Wind flute = new Wind();
    Instrument.tune(flute); // Upcasting
  }
}
```

What is interesting in this example is the **tune( )** method, which accepts an **Instrument** reference. However, in **Wind.main( )** the **tune( )** method is called by giving it a **Wind** reference. Given that Java is particular about type checking, it seems strange that a method that accepts one type will readily accept another type, until you realize that a **Wind** object is also an **Instrument** object.

Inside **tune( )**, the code works for **Instrument** and anything derived from **Instrument**, and the act of converting a **Wind** reference into an **Instrument** reference is called *upcasting*.

2.6 The final Keyword

Java's **final** keyword has slightly different meanings depending on the context, but in general it says "This cannot be changed." You might want to prevent changes for two reasons: design or efficiency. Final can be used in 3 places:

- data
- methods
- classes

2.6.1 Final Data

When applied to data, final means that piece of data is "constant". A constant can be:

- a compile-time constant that won't ever change (these sorts of constants must be primitives. A value must be given at the time of definition of such a constant)
- a value initialized at run time that you don not want changed

A field that is both **static** and **final** has only one piece of storage that cannot be changed.

With a primitive, **final** makes the value a constant, but with an object reference, **final** makes the reference a constant. Once the reference is initialized to an object, it can never be changed to point to another object. However, the object itself can be modified; Java does not provide a way to make any arbitrary object a constant. (You can, however, write your class so that objects have the effect of being constant.) This restriction includes arrays, which are also objects.

```
import java.util.*;
import static net.mindview.util.Print.*;

class Value {
  int i; // Package access
  public Value(int i) { this.i = i; }
}

public class FinalData {
  private static Random rand = new Random(47);
  private String id;
  public FinalData(String id) { this.id = id; }
  // Can be compile-time constants:
  private final int valueOne = 9;
  private static final int VALUE_TWO = 99;
  // Typical public constant:
  public static final int VALUE_THREE = 39;
  // Cannot be compile-time constants:
  private final int i4 = rand.nextInt(20);
  static final int INT_5 = rand.nextInt(20);
```

```
  private Value v1 = new Value(11);
  private final Value v2 = new Value(22);
  private static final Value VAL_3 = new Value(33);
  // Arrays:
  private final int[] a = { 1, 2, 3, 4, 5, 6 };
  public String toString() {
    return id + ": " + "i4 = " + i4 + ", INT_5 = " + INT_5;
  }
  public static void main(String[] args) {
    FinalData fd1 = new FinalData("fd1");
    //! fd1.valueOne++; // Error: can't change value
    fd1.v2.i++; // Object isn't constant!
    fd1.v1 = new Value(9); // OK -- not final
    for(int i = 0; i < fd1.a.length; i++)
      fd1.a[i]++; // Object isn't constant!
    //! fd1.v2 = new Value(0); // Error: Can't
    //! fd1.VAL_3 = new Value(1); // change reference
    //! fd1.a = new int[3];
    print(fd1);
    print("Creating new FinalData");
    FinalData fd2 = new FinalData("fd2");
    print(fd1);
    print(fd2);
  }
} /* Output:
fd1: i4 = 15, INT_5 = 18
Creating new FinalData
fd1: i4 = 15, INT_5 = 18
fd2: i4 = 13, INT_5 = 18
*///:~
```

2.6.2 Blank Finals

Java allows the creation of blank finals, which are fields that are declared as **final** but are NOT given an initialization value. In all cases, the blank final must be initialized before it is used, and the compiler ensures this.

You're forced to perform assignments to finals either with an expression at the point of definition of the field or in every constructor.

2.6.3 Final Arguments

Java allows you to make arguments final by declaring them as such in the argument list. This means that inside the method you cannot change what the argument reference points to.

```
class Gizmo {
  public void spin() {}
}

public class FinalArguments {
  void with(final Gizmo g) {
    //! g = new Gizmo(); // Illegal -- g is final
  }
  void without(Gizmo g) {
    g = new Gizmo(); // OK -- g not final
    g.spin();
  }
  // void f(final int i) { i++; } // Can't change
  // You can only read from a final primitive:
  int g(final int i) { return i + 1; }
  public static void main(String[] args) {
    FinalArguments bf = new FinalArguments();
    bf.without(null);
    bf.with(null);
  }
}
```

2.6.4 Final Methods

There are two reasons for final methods:

- The first is to put a "lock" on the method to prevent any inheriting class from changing its meaning. This is done for design reasons when you want to make sure that a method s behaviour is retained during inheritance and cannot be overridden.
- The second reason for final methods is efficiency. In earlier implementations of Java, if you made a method final, you allowed the compiler to turn any calls to that method into inline calls (this is no longer really needed so do not make a method final to help the compiler!).

Any **private** methods in a class are implicitly **final**. Because you can't access a private method, you can't override it. You can add the final specifier to a private method, but it doesn't give that method any extra meaning.

2.6.5 Final Classes

When you say that an entire class is **final** (by preceding its definition with the final keyword), you state that you don't want to inherit from this class or allow anyone else to do so. In other words, for some reason the design of your class is such that there is never a need to make any changes, or for safety or security

reasons you don't want subclassing.

Note that the fields of a **final** class can be final or not, as you choose. The same rules apply to final for fields regardless of whether the class is defined as final. However, because it prevents inheritance, all methods in a final class are implicitly final, since there's no way to override them.

2.7 Initialization and Class Loading

This is the sequence followed for initialization and class loading:

1. The first time an object of type X is created (or the first time a static method / static field of class X is executed / accessed), the Java interpreter locates the X.class file using the CLASSPATH variable.
2. X.class is loaded and static initialization is executed. It means static initialization takes place only once, when class X is loaded for the first time.
3. Storage for an object of type X is created in the heap and all primitives are initialized to default value.
4. Fields are initialized.
5. Constructors are executed (all variables are initialized before any method is executed, including constructors).

Further notes:

- Static fields are initialized in base and derived classes (in all of them)
- Non-static fields are initialized in base class and then the base class constructor is executed
- Non-static fields are initialized in the derived class and then the derived class constructor is executed

```
class Insect {
  private int i = 9;
  protected int j;
  Insect() {
    print("i = " + i + ", j = " + j);
    j = 39;
  }
  private static int x1 =
    printInit("static Insect.x1 initialized");
  static int printInit(String s) {
    print(s);
```

```
    return 47;
  }
}

public class Beetle extends Insect {
  private int k = printInit("Beetle.k initialized");
  public Beetle() {
    print("k = " + k);
    print("j = " + j);
  }
  private static int x2 =
    printInit("static Beetle.x2 initialized");
  public static void main(String[] args) {
    print("Beetle constructor");
    Beetle b = new Beetle();
  }
} /* Output:
static Insect.x1 initialized
static Beetle.x2 initialized
Beetle constructor
i = 9, j = 0
Beetle.k initialized
k = 47
j = 39
*///:~
```

Chapter 3

Polymorphism

3.1 Introduction

Polymorphism provides another dimension of separation of interface from implementation, to decouple *what* from *how*. Polymorphism allows improved code organization and readability as well as the creation of extensible programs that can be "grown" not only during the original creation of the project, but also when new features are desired.

Polymorphism allows to write a single method that takes the base class as its argument, and not any of the specific derived classes. So the code talks only to the base class and will work for all the derived classes.

Connecting a method call to a method body is called *binding*. When binding is performed before the program is run (by the compiler and linker, if there is one), it's called *early binding*. C compilers have only one kind of method call, and that's early binding.

Java works with *late binding* instead, which means that the binding occurs at run time, based on the type of object. Late binding is also called dynamic binding or runtime binding. When a language implements late binding, there must be some mechanism to determine the type of the object at run time and to call the appropriate method.

All method binding in Java uses late binding unless the method is static or final (private methods are implicitly final). This means that ordinarily you don't need to make any decisions about whether late binding will occur; it happens automatically.

Because all method binding in Java happens polymorphically via late binding, you can write your code to talk to the base class and know that all the derived-

class cases will work correctly using the same code.

3.2 What Does it Mean?

Polymorphism can be viewed by different perspectives.

3.2.1 Perspective 1 (Shape Example)

Because of late binding, even if a method is overridden in a derived class, the compiler knows if the base class method or the derived class method has to be used.

The upcast could occur in a statement as simple as:

```
Shape s = new Circle();
```

Suppose you call one of the base class methods (that have been overridden in the derived classes):

```
s.draw();
```

You might expect that **Shape**'s **draw( )** is called because this is, after all, a **Shape** reference-so how could the compiler know to do anything else? And yet the proper **Circle.draw( )** is called because of late binding (polymorphism).

3.2.2 Perspective 2 (Instrument Example)

Polymorphism allows the programmer to create a method which works with the base-class interface and this method will still work even when a derived-class is passed to it or when new derived classes are created later.

In the example below the **tune( )** method is blissfully ignorant of all the code changes that have happened around it, and yet it works correctly. This is exactly what polymorphism is supposed to provide.

```
import polymorphism.music.Note;
import static net.mindview.util.Print.*;

class Instrument {
  void play(Note n) { print("Instrument.play() " + n); }
  String what() { return "Instrument"; }
  void adjust() { print("Adjusting Instrument"); }
}

class Wind extends Instrument {
  void play(Note n) { print("Wind.play() " + n); }
  String what() { return "Wind"; }
  void adjust() { print("Adjusting Wind"); }
```

```
}

class Percussion extends Instrument {
  void play(Note n) { print("Percussion.play() " + n); }
  String what() { return "Percussion"; }
  void adjust() { print("Adjusting Percussion"); }
}

class Stringed extends Instrument {
  void play(Note n) { print("Stringed.play() " + n); }
  String what() { return "Stringed"; }
  void adjust() { print("Adjusting Stringed"); }
}

class Brass extends Wind {
  void play(Note n) { print("Brass.play() " + n); }
  void adjust() { print("Adjusting Brass"); }
}

class Woodwind extends Wind {
  void play(Note n) { print("Woodwind.play() " + n); }
  String what() { return "Woodwind"; }
}

public class Music3 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  public static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
  }
  public static void tuneAll(Instrument[] e) {
    for(Instrument i : e)
      tune(i);
  }
  public static void main(String[] args) {
    // Upcasting during addition to the array:
    Instrument[] orchestra = {
      new Wind(),
      new Percussion(),
      new Stringed(),
      new Brass(),
      new Woodwind()
    };
    tuneAll(orchestra);
  }
```

```
} /* Output:
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
*///:~
```

3.3 Pitfall 1 - Overriding private Methods

Remember that a **private** method is automatically final and is also hidden from the derived class.
The result of this is that only non-private methods may be overridden, but you should watch out for the appearance of overriding **private** methods, which generates no compiler warnings, but doesn't do what you might expect. To be clear, you should use a different name from a private base-class method in your derived class.

```
import static net.mindview.util.Print.*;

public class PrivateOverride {
  private void f() { print("private f()"); }
  public static void main(String[] args) {
    PrivateOverride po = new Derived();
    po.f();
  }
}

class Derived extends PrivateOverride {
  public void f() { print("public f()"); }
} /* Output:
private f()
*///:~
```

3.4 Pitfall 2 - Fields and static Methods

Only ordinary method calls can be polymorphic.
If you access a field directly, that access will be resolved at compile time, as the following example demonstrates:

```
class Super {
  public int field = 0;
  public int getField() { return field; }
}
```

```
class Sub extends Super {
  public int field = 1;
  public int getField() { return field; }
  public int getSuperField() { return super.field; }
}

public class FieldAccess {
  public static void main(String[] args) {
    Super sup = new Sub(); // Upcast
    System.out.println("sup.field = " + sup.field +
      ", sup.getField() = " + sup.getField());
    Sub sub = new Sub();
    System.out.println("sub.field = " +
      sub.field + ", sub.getField() = " +
      sub.getField() +
      ", sub.getSuperField() = " +
      sub.getSuperField());
  }
} /* Output:
sup.field = 0, sup.getField() = 1
sub.field = 1, sub.getField() = 1, sub.getSuperField() = 0
*///:~
```

Furthermore if a method is **static**, it doesn't behave polymorphically:

```
package polymorphism;

class StaticSuper {
  public static String staticGet() {
    return "Base staticGet()";
  }
  public String dynamicGet() {
    return "Base dynamicGet()";
  }
}

class StaticSub extends StaticSuper {
  public static String staticGet() {
    return "Derived staticGet()";
  }
  public String dynamicGet() {
    return "Derived dynamicGet()";
  }
}

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

```
    StaticSuper sup = new StaticSub(); // Upcast
    System.out.println(sup.staticGet());
    System.out.println(sup.dynamicGet());
  }
} /* Output:
Base staticGet()
Derived dynamicGet()
*///:~
```

3.5 Constructors and Polymorphism

Even though constructors are not polymorphic (they're actually static methods, but the static declaration is implicit), it's important to understand the way constructors work in complex hierarchies and with polymorphism.

A constructor for the base class is always called during the construction process for a derived class, chaining up the inheritance hierarchy so that a constructor for every base class is called.

The steps are:

1. The base-class constructor is called. This step is repeated recursively such that the root of the hierarchy is constructed first, followed by the next derived class, etc., until the most-derived class is reached.
2. Member initializers are called in the order of declaration.
3. The body of the derived-class constructor is called.

Example:

```
import static net.mindview.util.Print.*;

class Meal {
  Meal() { print("Meal()"); }
}

class Bread {
  Bread() { print("Bread()"); }
}

class Cheese {
  Cheese() { print("Cheese()"); }
}

class Lettuce {
  Lettuce() { print("Lettuce()"); }
```

```
}

class Lunch extends Meal {
  Lunch() { print("Lunch()"); }
}

class PortableLunch extends Lunch {
  PortableLunch() { print("PortableLunch()");}
}

public class Sandwich extends PortableLunch {
  private Bread b = new Bread();
  private Cheese c = new Cheese();
  private Lettuce l = new Lettuce();
  public Sandwich() { print("Sandwich()"); }
  public static void main(String[] args) {
    new Sandwich();
  }
} /* Output:
Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Sandwich()
*///:~
```

3.5.1 Polymorphic Methods Inside Constructors

The hierarchy of constructor calls brings up an interesting dilemma. What happens if you're inside a constructor and you call a dynamically-bound method of the object being constructed?

If you call a dynamically-bound method inside a constructor, the overridden definition for that method is used. However, the effect of this call can be rather unexpected because the overridden method will be called before the object is fully constructed. This can conceal some difficult-to-find bugs.

As a result, a good guideline for constructors is, "Do as little as possible to set the object into a good state, and if you can possibly avoid it, don't call any other methods in this class." The only safe methods to call inside a constructor are those that are final in the base class.

3.6 Covariant Return Types

Java SE5 adds *covariant return types*, which means that an overridden method in a derived class can return a type derived from the type returned by the base-class method:

```
class Grain {
  public String toString() { return "Grain"; }
}

class Wheat extends Grain {
  public String toString() { return "Wheat"; }
}

class Mill {
  Grain process() { return new Grain(); }
}

class WheatMill extends Mill {
  Wheat process() { return new Wheat(); }
}

public class CovariantReturn {
  public static void main(String[] args) {
    Mill m = new Mill();
    Grain g = m.process();
    System.out.println(g);
    m = new WheatMill();
    g = m.process();
    System.out.println(g);
  }
} /* Output:
Grain
Wheat
*///:~
```

3.7 Designing With Inheritance

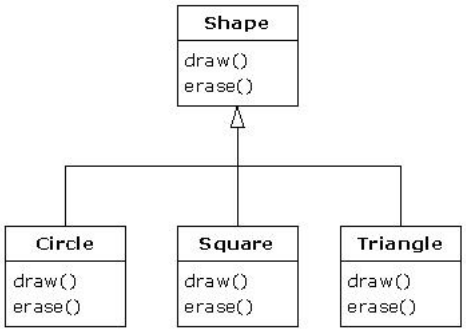
Once you learn about polymorphism, it can seem that everything ought to be inherited, because polymorphism is such a clever tool. This can burden your designs; in fact, if you choose inheritance first when you're using an existing class to make a new class, things can become needlessly complicated.

A better approach is to choose composition first, especially when it's not obvious which one you should use.

A general guideline is "Use inheritance to express differences in behaviour, and fields to express variations in state."

3.7.1 Substitution vs. Extension

It would seem that the cleanest way to create an inheritance hierarchy is to take the "pure" approach. That is, only methods that have been established in the base class are overridden in the derived class, as seen in this diagram:



This can be called a pure "is-a" relationship because the interface of a class establishes what it is. Inheritance guarantees that any derived class will have the interface of the base class and nothing less. If you follow this diagram, derived classes will also have no more than the base-class interface.
This can be thought of as pure substitution, because derived class objects can be perfectly substituted for the base class, and you never need to know any extra information about the subclasses when you're using them.

But sometimes extending the interface (which, unfortunately, the keyword **extends** seems to encourage) is the perfect solution to a particular problem. This can be termed an *"is-like-a"* relationship, because the derived class is like the base class-it has the same fundamental interface-but it has other features that require additional methods to implement.

While this is also a useful and sensible approach (depending on the situation), it has a drawback. The extended part of the interface in the derived class is not available from the base class.

Chapter 4

Interfaces

4.1 Abstract Classes and Methods

You create an abstract class when you want to manipulate a set of classes through its common interface.

Java provides a mechanism for doing this called the *abstract method*. This is a method that is incomplete; it has only a declaration and no method body. Here is the syntax for an abstract method declaration.

```
abstract void f( );
```

A class containing abstract methods is called an abstract class. If a class contains one or more abstract methods, the class itself must be qualified as abstract. (Otherwise, the compiler gives you an error message.)

If you inherit from an abstract class and you want to make objects of the new type, you must provide method definitions for all the abstract methods in the base class. If you don't (and you may choose not to), then the derived class is also abstract, and the compiler will force you to qualify that class with the abstract keyword.

It's possible to make a class abstract without including any abstract methods. This is useful when you've got a class in which it doesn't make sense to have any abstract methods, and yet you want to prevent any instances of that class.

It's helpful to create abstract classes and methods because they make the abstractness of a class explicit, and tell both the user and the compiler how it was intended to be used.

4.2 Interfaces

The **interface** keyword takes the concept of abstractness one step further. The **abstract** keyword allows you to create one or more undefined methods in a class-you provide part of the interface without providing a corresponding implementation. The implementation is provided by inheritors.

The **interface** keyword produces a completely abstract class, one that provides no implementation at all. It allows the creator to determine method names, argument lists, and return types, but no method bodies. An interface provides only a form, but no implementation.
An interface says, "All classes that implement this particular interface will look like this." Thus, any code that uses a particular interface knows what methods might be called for that interface, and that's all. So the interface is used to establish a "protocol" between classes.

To create an interface, use the **interface** keyword instead of the **class** keyword. As with a class, you can add the **public** keyword before the **interface** keyword (but only if that interface is defined in a file of the same name). If you leave off the **public** keyword, you get package access, so the interface is only usable within the same package. An interface can also contain fields, but these are implicitly static and final.

To make a class that conforms to a particular interface (or group of interfaces), use the **implements** keyword, which says, "The interface is what it looks like, but now I am going to say how it works."

You can see from the Woodwind and Brass classes that once you've implemented an interface, that implementation becomes an ordinary class that can be extended in the regular way.

You can choose to explicitly declare the methods in an interface as **public**, but they are public even if you don't say it. So when you implement an interface, the methods from the interface must be defined as public.

4.3 Complete Decoupling

Whenever a method works with a class instead of an interface, you are limited to using that class or its subclasses. If you would like to apply the method to a class that isn't in that hierarchy, you're out of luck. An interface relaxes this constraint considerably. As a result, it allows you to write more reusable code.

4.4 Multiple Inheritance in Java

Because an interface has no implementation at all-that is, there is no storage associated with an interface-there's nothing to prevent many interfaces from being combined.

If you do inherit from a non-interface, you can inherit from only one. All the rest of the base elements must be interfaces. You place all the interface names after the implements keyword and separate them with commas. You can have as many interfaces as you want. You can upcast to each interface, because each interface is an independent type.

```
interface CanFight {
  void fight();
}

interface CanSwim {
  void swim();
}

interface CanFly {
  void fly();
}

class ActionCharacter {
  public void fight() {}
}

class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
  public void swim() {}
  public void fly() {}
}

public class Adventure {
  public static void t(CanFight x) { x.fight(); }
  public static void u(CanSwim x) { x.swim(); }
  public static void v(CanFly x) { x.fly(); }
  public static void w(ActionCharacter x) { x.fight(); }
  public static void main(String[] args) {
    Hero h = new Hero();
    t(h); // Treat it as a CanFight
    u(h); // Treat it as a CanSwim
    v(h); // Treat it as a CanFly
    w(h); // Treat it as an ActionCharacter
  }
```

```
} ///:~
```

You can see that Hero combines the concrete class **ActionCharacter** with the interfaces **CanFight**, **CanSwim**, and **CanFly**. When you combine a concrete class with interfaces this way, the concrete class must come first, then the interfaces. (The compiler gives an error otherwise.)

Should you use an interface or an abstract class? If it's possible to create your base class without any method definitions or member variables, you should always prefer interfaces to abstract classes.

4.5 Extending an Interface With Inheritance

You can easily add new method declarations to an interface by using inheritance, and you can also combine several interfaces into a new interface with inheritance. In both cases you get a new interface, as seen in this example:

```
interface Monster {
  void menace();
}

interface DangerousMonster extends Monster {
  void destroy();
}

interface Lethal {
  void kill();
}

class DragonZilla implements DangerousMonster {
  public void menace() {}
  public void destroy() {}
}

interface Vampire extends DangerousMonster, Lethal {
  void drinkBlood();
}

class VeryBadVampire implements Vampire {
  public void menace() {}
  public void destroy() {}
  public void kill() {}
  public void drinkBlood() {}
}

public class HorrorShow {
```

```
  static void u(Monster b) { b.menace(); }
  static void v(DangerousMonster d) {
    d.menace();
    d.destroy();
  }
  static void w(Lethal l) { l.kill(); }
  public static void main(String[] args) {
    DangerousMonster barney = new DragonZilla();
    u(barney);
    v(barney);
    Vampire vlad = new VeryBadVampire();
    u(vlad);
    v(vlad);
    w(vlad);
  }
} ///:~
```

The syntax used in **Vampire** works only when inheriting interfaces. Normally, you can use extends with only a single class, but extends can refer to multiple base interfaces when building a new interface. As you can see, the interface names are simply separated with commas.

4.6 Fields in Interfaces

Because any fields you put into an interface are automatically static and final, the interface is a convenient tool for creating groups of constant values. Before Java SE5, this was the only way to produce the same effect as an **enum**.

With Java SE5, you now have the much more powerful and flexible enum keyword, so it rarely makes sense to use interfaces for constants anymore.

Fields defined in interfaces cannot be "blank finals" but they can be initialized with non-constant expressions. For example:

```
import java.util.*;

public interface RandVals {
  Random RAND = new Random(47);
  int RANDOM_INT = RAND.nextInt(10);
  long RANDOM_LONG = RAND.nextLong() * 10;
  float RANDOM_FLOAT = RAND.nextLong() * 10;
  double RANDOM_DOUBLE = RAND.nextDouble() * 10;
} ///:~
```

4.7 Interfaces and Factories

An interface is intended to be a gateway to multiple implementations, and a typical way to produce objects that fit the interface is the *Factory Method design pattern*.
Instead of calling a constructor directly, you call a creation method on a factory object which produces an implementation of the interface; this way, in theory, your code is completely isolated from the implementation of the interface, thus making it possible to transparently swap one implementation for another.
Here's a demonstration showing the structure of the Factory Method:

```
import static net.mindview.util.Print.*;

interface Service {
  void method1();
  void method2();
}

interface ServiceFactory {
  Service getService();
}

class Implementation1 implements Service {
  Implementation1() {} // Package access
  public void method1() {print("Implementation1 method1");}
  public void method2() {print("Implementation1 method2");}
}

class Implementation1Factory implements ServiceFactory {
  public Service getService() {
    return new Implementation1();
  }
}

class Implementation2 implements Service {
  Implementation2() {} // Package access
  public void method1() {print("Implementation2 method1");}
  public void method2() {print("Implementation2 method2");}
}

class Implementation2Factory implements ServiceFactory {
  public Service getService() {
    return new Implementation2();
  }
}
```

```
public class Factories {
  public static void serviceConsumer(ServiceFactory fact) {
    Service s = fact.getService();
    s.method1();
    s.method2();
  }
  public static void main(String[] args) {
    serviceConsumer(new Implementation1Factory());
    // Implementations are completely interchangeable:
    serviceConsumer(new Implementation2Factory());
  }
} /* Output:
Implementation1 method1
Implementation1 method2
Implementation2 method1
Implementation2 method2
*///:~
```

Without the Factory Method, your code would somewhere have to specify the exact type of Service being created, so that it could call the appropriate constructor.

Why would you want to add this extra level of indirection? One common reason is to create a framework. Suppose you are creating a system to play games; for example, to play both chess and checkers on the same board:

```
import static net.mindview.util.Print.*;

interface Game { boolean move(); }
interface GameFactory { Game getGame(); }

class Checkers implements Game {
  private int moves = 0;
  private static final int MOVES = 3;
  public boolean move() {
    print("Checkers move " + moves);
    return ++moves != MOVES;
  }
}

class CheckersFactory implements GameFactory {
  public Game getGame() { return new Checkers(); }
}

class Chess implements Game {
  private int moves = 0;
  private static final int MOVES = 4;
```

```
  public boolean move() {
    print("Chess move " + moves);
    return ++moves != MOVES;
  }
}

class ChessFactory implements GameFactory {
  public Game getGame() { return new Chess(); }
}

public class Games {
  public static void playGame(GameFactory factory) {
    Game s = factory.getGame();
    while(s.move())
      ;
  }
  public static void main(String[] args) {
    playGame(new CheckersFactory());
    playGame(new ChessFactory());
  }
} /* Output:
Checkers move 0
Checkers move 1
Checkers move 2
Chess move 0
Chess move 1
Chess move 2
Chess move 3
*///:~
```

If the Games class represents a complex piece of code, this approach allows you to reuse that code with different types of games. You can imagine more elaborate games that can benefit from this pattern. In the next chapter, you'll see a more elegant way to implement the factories using anonymous inner classes.

4.8 Interfaces Summary

An appropriate guideline is to prefer classes to interfaces. Start with classes, and if it becomes clear that interfaces are necessary, then refactor. Interfaces are a great tool, but they can easily be overused.

Chapter 5

Inner Classes

The inner class is a valuable feature because it allows you to group classes that logically belong together and to control the visibility of one within the other. However, it's important to understand that inner classes are distinctly different from composition.

5.1 Basic Form

An inner class in its simplest form is a class inside the definition of another class (just using the keyword **class** as usual); example:

```
public class Outer {
        public int x;
        public void f(){}

        class Inner{
                int y;
                void z() {}
        } // End of Inner
} // End of Outer
```

The inner class can access all members of the outer class, even the private ones.

If you use the inner class within the outer class, you can handle the inner class as a normal class (i.e. create an object of the inner class and use it as usual).

If you need to produce the reference to the outer-class object, you name the outer class followed by a dot and **this**.

```
public class DotThis {
  void f() { System.out.println("DotThis.f()"); }
  public class Inner {
    public DotThis outer() {
```

```
      return DotThis.this;
      // A plain "this" would be Inner's "this"
    }
  }
  public Inner inner() { return new Inner(); }
  public static void main(String[] args) {
    DotThis dt = new DotThis();
    DotThis.Inner dti = dt.inner();
    dti.outer().f();
  }
} /* Output:
DotThis.f()
*///:~
```

Sometimes you want to tell some other object to create an object of one of its inner classes. To do this you must provide a reference to the other outer-class object in the new expression, using the **.new** syntax, like this:

```
public class DotNew {
  public class Inner {}
  public static void main(String[] args) {
    DotNew dn = new DotNew();
    DotNew.Inner dni = dn.new Inner();
  }
} ///:~
```

Inner classes can be **private** or **protected**, hence they can be used for implementation hiding; example:

```
class Parcel4 {
  private class PContents implements Contents {
    private int i = 11;
    public int value() { return i; }
  }
  protected class PDestination implements Destination {
    private String label;
    private PDestination(String whereTo) {
      label = whereTo;
    }
    public String readLabel() { return label; }
  }
  public Destination destination(String s) {
    return new PDestination(s);
  }
  public Contents contents() {
    return new PContents();
  }
}
```

```
public class TestParcel {
  public static void main(String[] args) {
    Parcel4 p = new Parcel4();
    Contents c = p.contents();
    Destination d = p.destination("Tasmania");
    // Illegal -- can't access private class:
    //! Parcel4.PContents pc = p.new PContents();
  }
} ///:~
```

5.2 Inner Classes in Methods and Scopes

Inner classes in a more complex form include:

- a class defined within a method
- a class defined within a scope inside a method
- an anonymous class implementing an interface
- an anonymous class extending a class

Inner classes can be static (or nested), in this case there is no connection between inner and outer class.

Chapter 6

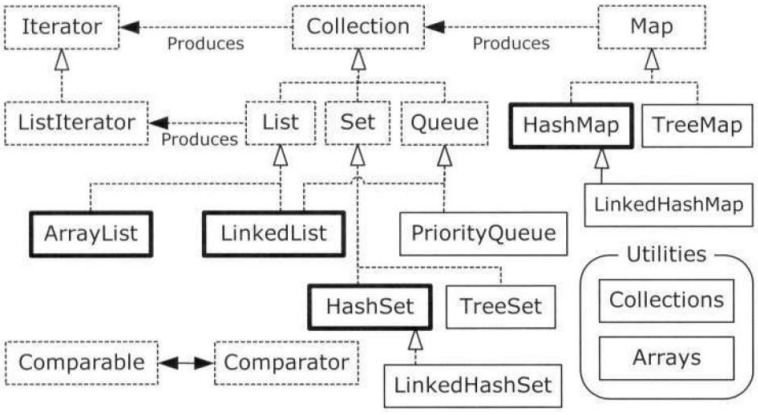
Holding Your Objects

Java has ways to hold objects (i.e. references to objects), most of them are part of the **java.util** library. There are container classes which can perform this task, the basic ones are:

- Array
- List
- Set
- Queue
- Map

The array is the simpler way to hold objects, but the main drawback is an array has a fixed size and this can be a limit at some point.

The taxonomy of the Java containers can be summarized as below



6.1 Generics and Type-safe Containers

Using the container **ArrayList**, it is possible to create a container with objects of a different type (e.g. Apples and Oranges). If generics are not used the compiler will only give a warning, but a runtime exception will be thrown if I try to fetch out the objects from the **ArrayList**.

```
package holding;
import java.util.*;

class Apple {
  private static long counter;
  private final long id = counter++;
  public long id() { return id; }
}

class Orange {}

public class ApplesAndOrangesWithoutGenerics {
  public static void main(String[] args) {
    ArrayList apples = new ArrayList();
    for(int i = 0; i < 3; i++)
      apples.add(new Apple());
    // Not prevented from adding an Orange to apples:
    apples.add(new Orange());
    for(int i = 0; i < apples.size(); i++)
      ((Apple)apples.get(i)).id();
      // Orange is detected only at run time
  }
} /* (Execute to see output) *///:~
```

To solve this problem generics can be used. The syntax to use generics is:

```
ArrayList<Class> name = new ArrayList<Class>()
```

where `<Class>` is the object the **ArrayList** has to hold. Using generics the Compiler will prevent to add to the list objects of a different type. Subclasses of the class `<Class>` can be added to the **ArrayList** too.

Example: to define an **ArrayList** intended to hold **Apple** objects, you say `ArrayList<Apple>` instead of just `ArrayList`; with generics, you're prevented, at compile time, from putting the wrong type of object into a container.

6.2 Basic concepts (keep in mind the taxonomy picture)

The idea of *holding your objects* has been implemented in Java mainly with 2 interfaces: **Collection** and **Map**.

- **Collection**: a sequence of individual elements with one or more rules applied to them. Example, a **List** holds the elements in the way they were inserted; a **Set** can not have duplicate elements; a **Queue** produces the elements in the order defined by a queuing discipline.
- **Map**: it is a group of key-value object pairs, it means the elements in the map can be accessed via a key (used to lookup the objects in the map). Example, an **ArrayList** allows to retrieve each element via the index number

The distinction between **Collection** and **Map** is based on the number of items that are held in each "slot" in the container.
The **Collection** category only holds one item in each slot. It includes:

- the **List**, which holds a group of items in a specified sequence
- the **Set**, which only allows the addition of one identical item
- the **Queue**, which only allows you to insert objects at one "end" of the container and remove objects from the other "end"

A **Map** holds two objects, a key and an associated value, in each slot.

Ideally all the code should be written to talk to these 2 basic interfaces, example:

```
List<Apple> apple = new ArrayList<Apple>()
```

so in the future only the implementation is changed if needed

```
List<Apple> apple = new LinkedList<Apple>()
```

Of course this approach will not always work when an implementation has additional features.

All Collections can be traversed using the foreach syntax. But there is a more flexible concept called an *Iterator* (see later).

6.3 Adding Groups of Elements

There are utility methods in both the **Arrays** and **Collections** classes in **java.util** that add groups of elements to a Collection:

- **Arrays.asList( )** takes either an array or a comma-separated list of elements (using varargs) and turns it into a List object.
- **Collections.addAll( )** takes a Collection object and either an array or a comma-separated list and adds the elements to the Collection.

Example:

```
import java.util.*;

public class AddingGroups {
  public static void main(String[] args) {
    Collection<Integer> collection = new ArrayList<Integer>(Arrays.asList(1, 2,
    3, 4, 5));
    Integer[] moreInts = { 6, 7, 8, 9, 10 };
    collection.addAll(Arrays.asList(moreInts));
    // Runs significantly faster, but you can't
    // construct a Collection this way:
    Collections.addAll(collection, 11, 12, 13, 14, 15);
    Collections.addAll(collection, moreInts);
    // Produces a list "backed by" an array:
    List<Integer> list = Arrays.asList(16, 17, 18, 19, 20);
    list.set(1, 99); // OK -- modify an element
    // list.add(21); // Runtime error because the
                     // underlying array cannot be resized.
  }
} ///:~
```

Collections.addAll( ) runs much faster, and it is just as easy to construct the **Collection** with no elements and then call **Collections.addAll( )**, so this is the preferred approach.
A limitation of **Arrays.asList( )** is that it takes a best guess about the resulting type of the **List**, and does not pay attention to what you are assigning it to. Sometimes this can cause a problem:

```
import java.util.*;

class Snow {}
class Powder extends Snow {}
class Light extends Powder {}
class Heavy extends Powder {}
class Crusty extends Snow {}
class Slush extends Snow {}

public class AsListInference {
  public static void main(String[] args) {
    List<Snow> snow1 = Arrays.asList(
      new Crusty(), new Slush(), new Powder());
```

```
    // Won't compile:
    // List<Snow> snow2 = Arrays.asList(
    //   new Light(), new Heavy());
    // Compiler says:
    // found    : java.util.List<Powder>
    // required: java.util.List<Snow>

    // Collections.addAll() doesn't get confused:
    List<Snow> snow3 = new ArrayList<Snow>();
    Collections.addAll(snow3, new Light(), new Heavy());

    // Give a hint using an
    // explicit type argument specification:
    List<Snow> snow4 = Arrays.<Snow>asList(
       new Light(), new Heavy());
  }
} ///:~
```

When trying to create **snow2**, **Arrays.asList( )** only has types of **Powder**, so it creates a **List<Powder>** rather than a **List<Snow>**, whereas **Collections.addAll( )** works fine because it knows from the first argument what the target type is.
As you can see from the creation of **snow4**, it is possible to insert a "hint" in the middle of **Arrays.asList( )**, to tell the compiler what the actual target type should be for the resulting **List** type produced by **Arrays.asList( )**. This is called *an explicit type argument specification.* **Maps** are more complex, as you will see, and the Java standard library does not provide any way to automatically initialize them, except from the contents of another **Map**.

6.4 Printing Containers

You must use **Arrays.toString( )** to produce a printable representation of an array, but the containers print nicely without any help.

```
import java.util.*;
import static net.mindview.util.Print.*;

public class PrintingContainers {
  static Collection fill(Collection<String> collection) {
    collection.add("rat");
    collection.add("cat");
    collection.add("dog");
    collection.add("dog");
    return collection;
  }
```

```
  static Map fill(Map<String,String> map) {
    map.put("rat", "Fuzzy");
    map.put("cat", "Rags");
    map.put("dog", "Bosco");
    map.put("dog", "Spot");
    return map;
  }
  public static void main(String[] args) {
    print(fill(new ArrayList<String>()));
    print(fill(new LinkedList<String>()));
    print(fill(new HashSet<String>()));
    print(fill(new TreeSet<String>()));
    print(fill(new LinkedHashSet<String>()));
    print(fill(new HashMap<String,String>()));
    print(fill(new TreeMap<String,String>()));
    print(fill(new LinkedHashMap<String,String>()));
  }
} /* Output:
[rat, cat, dog, dog]
[rat, cat, dog, dog]
[dog, cat, rat]
[cat, dog, rat]
[rat, cat, dog]
{dog=Spot, cat=Rags, rat=Fuzzy}
{cat=Rags, dog=Spot, rat=Fuzzy}
{rat=Fuzzy, cat=Rags, dog=Spot}
*///:~
```

6.5 List

Lists promise to maintain elements in a particular sequence. The **List** interface adds a number of methods to **Collection** that allow insertion and removal of elements in the middle of a List. There are two types of List:

- The basic **ArrayList**, which excels at randomly accessing elements, but is slower when inserting and removing elements in the middle of a List.
- The **LinkedList**, which provides optimal sequential access, with inexpensive insertions and deletions from the middle of the List. A **LinkedList** is relatively slow for random access, but it has a larger feature set than the ArrayList.

Some key characteristics of lists:

- Unlike an array, a List allows you to add elements after it has been created, or remove elements, and it resizes itself. That is its fundamental value: a modifiable sequence

- You can find out whether an object is in the list using the **contains( )** method
- If you want to remove an object, you can pass that object reference to the **remove( )** method.
- Also, if you have a reference to an object, you can discover the index number where that object is located in the List using **indexOf( )**
- It is possible to insert an element in the middle of the List using the method **add()**
- The **subList( )** method allows you to easily create a slice out of a larger list

There are more methods available. Check the standard Java documentation.

6.6 Iterator

An iterator is an object whose job is to move through a sequence and select each object in that sequence without the client programmer knowing or caring about the underlying structure of that sequence. In addition, an iterator is usually what is called a lightweight object: one that is cheap to create. For that reason, you will often find seemingly strange constraints for iterators; for example, the Java Iterator can move in only one direction. There is not much you can do with an Iterator except:

1. Ask a Collection to hand you an Iterator using a method called **iterator( )**. That Iterator will be ready to return the first element in the sequence
2. Get the next object in the sequence with **next( )**.
3. See if there are any more objects in the sequence with **hasNext( )**.
4. Remove the last element returned by the iterator with **remove( )**. An Iterator will remove the last element produced by **next( )**, which means you must call **next( )** before you call **remove( )**.

An iterator works with ArrayList, LinkedList, HastSet and TreeSet in the same way.

6.6.1 ListIterator

The **ListIterator** is a more powerful subtype of **Iterator** that is produced only by **List** classes. While **Iterator** can only move forward, **ListIterator** is bidirectional.
It can also produce the indexes of the next and previous elements relative to where the iterator is pointing in the list, and it can replace the last element that it visited using the **set( )** method. You can produce a **ListIterator** that

points to the beginning of the List by calling **listIterator( )**, and you can also create a **ListIterator** that starts out pointing to an index n in the list by calling **listIterator(n)**.

6.7 LinkedList

The **LinkedList** also implements the basic **List** interface like **ArrayList** does, but it performs certain operations (insertion and removal in the middle of the List) more efficiently than does **ArrayList**. Conversely, it is less efficient for random-access operations.
LinkedList also adds methods that allow it to be used as a stack, a Queue or a double-ended queue (deque).
Some of these methods are aliases or slight variations of each other, to produce names that are more familiar within the context of a particular usage (Queue, in particular). For example:

- **getFirst( )** and **element( )** are identical: they return the head (first element) of the list without removing it, and throw NoSuchElementException if the List is empty.
- **peek( )** is a slight variation of those two that returns null if the list is empty.
- **removeFirst( )** and **remove( )** are also identical they remove and return the head of the list, and throw NoSuchElementException for an empty list.
- **poll( )** is a slight variation that returns null if this list is empty.
- **addFirst( )** inserts an element at the beginning of the list.
- **offer( )** is the same as **add( )** and **addLast( )**. They all add an element to the tail (end) of a list.
- **removeLast( )** removes and returns the last element of the list.

6.7.1 Stack

A stack is sometimes referred to as a "last-in, first-out" (LIFO) container. **LinkedList** has methods that directly implement stack functionality, so you can also just use a **LinkedList** rather than making a stack class. However, a stack class can sometimes tell the story better:

```
package net.mindview.util;
import java.util.LinkedList;
public class Stack<T> {
        private LinkedList<T> storage = new LinkedList<T>();
        public void push(T v) { storage.addFirst(v); }
        public T peek() { return storage.getFirst(); }
```

```
        public T pop() { return storage.removeFirst(); }
        public boolean empty() { return storage.isEmpty(); }
        public String toString() { return storage.toString(); }
} ///:~
```

This introduces the simplest possible example of a class definition using generics. The <T>after the class name tells the compiler that this will be a parameterized type, and that the type parameter –the one that will be substituted with a real type when the class is used –is T. Basically, this says, "We are defining a Stack that holds objects of type T."
The Stack is implemented using a **LinkedList**, and the **LinkedList** is also told that it is holding type T. Notice that **push( )** takes an object of type T, while **peek( )** and **pop( )** return an object of type T. The **peek( )** method provides you with the top element without removing it from the top of the stack, while **pop( )** removes and returns the top element.
Here is a simple demonstration of this new Stack class:

```
import net.mindview.util.*;

public class StackTest {
  public static void main(String[] args) {
    Stack<String> stack = new Stack<String>();
    for(String s : "My dog has fleas".split(" "))
      stack.push(s);
    while(!stack.empty())
      System.out.print(stack.pop() + " ");
  }
} /* Output:
fleas has dog My
*///:~
```

If you want to use this **Stack** class in your own code, you will need to fully specify the package (or change the name of the class) when you create one; otherwise, you will probably collide with the Stack in the java.util package. For example, if we import java.util.* into the above example, we must use package names in order to prevent collisions.

6.8 Set

A **Set** refuses to hold more than one instance of each object value. If you try to add more than one instance of an equivalent object, the **Set** prevents duplication. The most common use for a **Set** is to test for membership, so that you can easily ask whether an object is in a **Set**. Because of this, lookup is typically the most important operation for a **Set**, so you will usually choose a **HashSet** implementation, which is optimized for rapid lookup.
A **Set** determines membership based on the "value" of an object.

```
import java.util.*;

public class SetOfInteger {
  public static void main(String[] args) {
    Random rand = new Random(47);
    Set<Integer> intset = new HashSet<Integer>();
    for(int i = 0; i < 10000; i++)
      intset.add(rand.nextInt(30));
    System.out.println(intset);
  }
} /* Output:
[15, 8, 23, 16, 7, 22, 9, 21, 6, 1, 29, 14, 24, 4, 19, 26, 11, 18, 3, 12, 27,
17, 2, 13, 28, 20, 25, 10, 5, 0]
*///:~
```

You will also notice that the output is in no discernible order. This is because a **HashSet** uses hashing for speed–hashing is covered in the Containers in Depth chapter. The order maintained by a **HashSet** is different from a **TreeSet** or a **LinkedHashSet**, since each implementation has a different way of storing elements.

TreeSet keeps elements sorted into a red-black tree data structure, whereas **HashSet** uses the hashing function. **LinkedHashSet** also uses hashing for lookup speed, but appears to maintain elements in insertion order using a linked list.

If you want the results to be sorted, one approach is to use a TreeSet instead of a HashSet:

```
package holding;

import java.util.*;

public class SortedSetOfInteger {
  public static void main(String[] args) {
    Random rand = new Random(47);
    SortedSet<Integer> intset = new TreeSet<Integer>();
    for(int i = 0; i < 10000; i++)
      intset.add(rand.nextInt(30));
    System.out.println(intset);
  }
} /* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29]
*///:~
```

Methods **contains()** and **containsAll()**, can be used to test if an element belongs to a **Set** or if a subset of elements belongs to a Set. More methods are available; check the SDK documentation.

6.9 Map

A **Map** is a container with a structure based on an index and a value associated to each index. Both the index and the values can be associated to any primitive data type or classes.

Example, a Map of integers.

```
package holding;

import java.util.*;

public class Statistics {
  public static void main(String[] args) {
    Random rand = new Random(47);
    Map<Integer,Integer> m =
      new HashMap<Integer,Integer>();
    for(int i = 0; i < 10000; i++) {
      // Produce a number between 0 and 20:
      int r = rand.nextInt(20);
      Integer freq = m.get(r);
      m.put(r, freq == null ? 1 : freq + 1);
    }
    System.out.println(m);
  }
}
```

Or a map of String and Pets.

```
package holding;

import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class PetMap {
        public static void main(String[] args) {
                Map<String,Pet> petMap = new HashMap<String,Pet>();
                petMap.put("My Cat", new Cat("Molly"));
                petMap.put("My Dog", new Dog("Ginger"));
                petMap.put("My Hamster", new Hamster("Bosco"));
                print(petMap);
                Pet dog = petMap.get("My Dog");
                print(dog);
                print(petMap.containsKey("My Dog"));
                print(petMap.containsValue(dog));
        }
}
```

Maps, like **Arrays** and **Collections**, can easily be expanded to multiple dimensions; you simply make a **Map** whose values are **Maps** (and the values of those **Maps** can be other containers, even other **Maps**). Thus, it is quite easy to combine containers to quickly produce powerful data structures. For example, suppose you are keeping track of people who have multiple pets-all you need is a Map <Person>, List <Pet>>:

```
//: holding/MapOfList.java
package holding;

import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class MapOfList {
        public static Map<Person, List<? extends Pet>>
        petPeople = new HashMap<Person, List<? extends Pet>>();
        static {
                petPeople.put(new Person("Dawn"),
                                Arrays.asList(new Cymric("Molly"),new Mutt("Spot
                petPeople.put(new Person("Kate"),
                                Arrays.asList(new Cat("Shackleton"),
                                                new Cat("Elsie May"), new Dog("M
                petPeople.put(new Person("Marilyn"),
                                Arrays.asList(
                                                new Pug("Louie aka Louis Snorkel
                                                new Cat("Stanford aka Stinky el
                                                new Cat("Pinkola")));
                petPeople.put(new Person("Luke"),
                                Arrays.asList(new Rat("Fuzzy"), new Rat("Fizzy")
                petPeople.put(new Person("Isaac"),
                                Arrays.asList(new Rat("Freckly")));
        }
        public static void main(String[] args) {
                print("People: " + petPeople.keySet());
                print("Pets: " + petPeople.values());
                for(Person person : petPeople.keySet()) {
                        print(person + " has:");
                        for(Pet pet : petPeople.get(person))
                                print("    " + pet);
                }
        }
}
```

A **Map** can return a **Set** of its keys, a **Collection** of its values, or a **Set** of its pairs. The **keySet( )** method produces a **Set** of all the keys.

6.10 Queue

A **queue** is typically a "first-in, first-out" (FIFO) container. That is, you put things in at one end and pull them out at the other, and the order in which you put them in will be the same order in which they come out. Queues are commonly used as a way to reliably transfer objects from one area of a program to another. **LinkedList** has methods to support queue behavior and it implements the **Queue** interface, so a **LinkedList** can be used as a **Queue** implementation.

```
import java.util.*;

public class QueueDemo {
  public static void printQ(Queue queue) {
    while(queue.peek() != null)
      System.out.print(queue.remove() + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    Queue<Integer> queue = new LinkedList<Integer>();
    Random rand = new Random(47);
    for(int i = 0; i < 10; i++)
      queue.offer(rand.nextInt(i + 10));
    printQ(queue);
    Queue<Character> qc = new LinkedList<Character>();
    for(char c : "Brontosaurus".toCharArray())
      qc.offer(c);
    printQ(qc);
  }
}
```

Some of the **Queue** specific methods are:

- **offer( )** it inserts an element at the tail of the queue if it can, or returns false
- **peek( )** returns the head of the queue without removing it and returns null if the queue is empty
- **element( )** returns the head of the queue without removing it and throws **NoSuchElementException** if the queue is empty
- **poll( )** removes and returns the head of the queue and returns null if the queue is empty
- **remove( )** removes and returns the head of the queue and throws **NoSuchElementException** if the queue is empty

The **Queue** interface narrows access to the methods of **LinkedList** so that only the appropriate methods are available, and you are thus less tempted

to use **LinkedList** methods (here, you could actually cast queue back to a **LinkedList**, but you are at least discouraged from doing so).

6.10.1 Priority Queue

A *priority queue* says that element that goes next is the one with the greatest need (the highest priority). The **PriorityQueue** was added in Java SE5 to provide an automatic implementation for this behavior. When you **offer( )** an object onto a **PriorityQueue**, *that object is sorted into the queue.* The default sorting uses the natural order of the objects in the queue, but you can modify the order by providing your own **Comparator**. The **PriorityQueue** ensures that when you call **peek( )**, **poll( )** or r**emove( )**, the element you get will be the one with the highest priority.

Integer, **String** and **Character** work with **PriorityQueue** because these classes already have natural ordering built in. If you want you use your own class in a **PriorityQueue**, you must include additional functionality to produce natural ordering, or provide your own **Comparator**.

6.11 Foreach and Iterators

So far, the foreach syntax has been primarily used with arrays, but it also works with any **Collection** object. Example:

```
import java.util.*;

public class ForEachCollections {
        public static void main(String[] args) {
                Collection<String> cs = new LinkedList<String>();
                Collections.addAll(cs,
                                "Take the long way home".split(" "));
                for(String s : cs)
                        System.out.print("'" + s + "' ");
        }
}
```

Since **cs** is a **Collection**, this code shows that working with **foreach** is a characteristic of all **Collection** objects. The reason that this works is that Java SE5 introduced a new interface called **Iterable** which contains an **iterator( )** method to produce an **Iterator**, and the **Iterable** interface is what **foreach** uses to move through a sequence. So if you create any class that implements **Iterable**, you can use it in a **foreach** statement:

```
import java.util.*;

public class IterableClass implements Iterable<String> {
        protected String[] words = ("And that is how " +
                        "we know the Earth to be banana-shaped.").split(" ");
```

```
        public Iterator<String> iterator() {
                return new Iterator<String>() {
                        private int index = 0;
                        public boolean hasNext() {
                                return index < words.length;
                        }
                        public String next() { return words[index++]; }
                        public void remove() { // Not implemented
                                throw new UnsupportedOperationException();
                        }
                };
        }
        public static void main(String[] args) {
                for(String s : new IterableClass())
                        System.out.print(s + " ");
        }
}
```

The **iterator( )** method returns an instance of an anonymous inner implementation of **Iterator¡String¿** which delivers each word in the array. In **main( )**, you can see that **IterableClass** does indeed work in a foreach statement.

NOTE: A foreach statement works with an array or anything IterableClass, but that doesn not mean that an array is automatically an IterableClass, nor is there any autoboxing that takes place:

```
import java.util.*;

public class ArrayIsNotIterable {
        static <T> void test(Iterable<T> ib) {
                for(T t : ib)
                        System.out.print(t + " ");
        }
        public static void main(String[] args) {
                test(Arrays.asList(1, 2, 3));
                String[] strings = { "A", "B", "C" };
                // An array works in foreach, but it's not Iterable:
                //! test(strings);
                // You must explicitly convert it to an Iterable:
                test(Arrays.asList(strings));
        }
}
```

Trying to pass an array as an **Iterable** argument fails. There is no automatic conversion to an **Iterable**; you must do it by hand.

6.12 Reverse Iterator

Suppose you wouldd like to choose whether to iterate through a list of words in either a forward or reverse direction. If you simply inherit from the class and override the **iterator( )** method, you replace the existing method and you don not get a choice. A solution is to add a method that produces an **Iterable** object which can then be used in the foreach statement. Example:

```
import java.util.*;

class ReversibleArrayList<T> extends ArrayList<T> {
        public ReversibleArrayList(Collection<T> c) { super(c); }
        public Iterable<T> reversed() {
                return new Iterable<T>() {
                        public Iterator<T> iterator() {
                                return new Iterator<T>() {
                                        int current = size() - 1;
                                        public boolean hasNext() { return curren
                                        public T next() { return get(current--);
                                        public void remove() { // Not implemente
                                                throw new UnsupportedOperationEx
                                        }
                                };
                        }
                };
        }
}

public class AdapterMethodIdiom {
        public static void main(String[] args) {
                ReversibleArrayList<String> ral =
                        new ReversibleArrayList<String>(
                                Arrays.asList("To be or not to be".split
                // Grabs the ordinary iterator via iterator():
                for(String s : ral)
                        System.out.print(s + " ");
                System.out.println();
                // Hand it the Iterable of your choice
                for(String s : ral.reversed())
                        System.out.print(s + " ");
        }
}
/* Output:
To be or not to be
be to not or be To
 *///:~
```

If you simply put the **ral** object in the foreach statement, you get the (default) forward iterator. But if you call **reversed( )** on the object, it produces different behavior.

Chapter 7

Error Handling With Exceptions

The ideal time to catch an error is at compile time, before you even try to run the program. However, not all errors can be detected at compile time. The rest of the problems must be handled at run time through some formality that allows the originator of the error to pass appropriate information to a recipient who will know how to handle the difficulty properly.
The "formality" implemented in Java is based on exceptions. The word "exception" is meant in the sense of "I take exception to that." At the point where the problem occurs, you might not know what to do with it, but you do know that you can not just continue on merrily; you must stop, and somebody, somewhere, must figure out what to do. But you don not have enough information in the current context to fix the problem. So you hand the problem out to a higher context where someone is qualified to make the proper decision.
The other rather significant benefit of exceptions is that they tend to reduce the complexity of error-handling code. With exceptions, you no longer need to check for errors at the point of the method call, since the exception will guarantee that someone catches it. You only need to handle the problem in one place, in the so-called exception handler. This saves you code, and it separates the code that describes what you want to do during normal execution from the code that is executed when things go wrong.

7.1 Basic Exceptions

An *exceptional condition* is a problem that prevents the continuation of the current method or scope. With an exceptional condition, you cannot continue processing because you don not have the information necessary to deal with the problem in the current context. All you can do is jump out of the current context and relegate that problem to a higher context. This is what happens when you throw an exception.

When you throw an exception, several things happen:

- First, the exception object is created in the same way that any Java object is created: on the heap, with **new**
- Then the current path of execution (the one you could not continue) is stopped and the reference for the exception object is ejected from the current context.
- At this point the exception-handling mechanism takes over and begins to look for an appropriate place to continue executing the program. This appropriate place is the exception handler, whose job is to recover from the problem so the program can either try another tack or just continue.

An example of throwing an exception:

```
if(t == null)
throw new NullPointerException();
```

7.1.1 Exception Arguments

As with any object in Java, you always create exceptions on the heap using new, which allocates storage and calls a constructor. There are two constructors in all standard exceptions: The first is the default constructor, and the second takes a string argument so that you can place pertinent information in the exception:

```
throw new NullPointerException("t = null");
```

After creating an exception object with **new**, you give the resulting reference to **throw**. A simplistic way to think about exception handling is as a different kind of return mechanism.
You can throw any type of **Throwable**, which is the exception root class. Typically, you will throw a different class of exception for each different type of error. The information about the error is represented both inside the exception object and implicitly in the name of the exception class.

7.2 Catching An Exception

To see how an exception is caught, you must first understand the concept of a guarded region. This is a section of code that might produce exceptions and is followed by the code to handle those exceptions.

7.2.1 The Try Block

If you are inside a method and you throw an exception (or another method that you call within this method throws an exception), that method will exit in the process of throwing. If you don not want a **throw** to exit the method, you can set up a special block within that method to capture the exception. This is called

the try block because you "try" your various method calls there. The **try** block is an ordinary scope preceded by the keyword **try**:

```
try {
// Code that might generate exceptions
}
```

7.2.2 Exception Handlers

Of course, the thrown exception must end up someplace. This "place" is the *exception handler*, and there is one for every exception type you want to catch. Exception handlers immediately follow the try block and are denoted by the keyword **catch**:

```
try {
// Code that might generate exceptions
} catch(Type1 id1) {
// Handle exceptions of Type1
} catch(Type2 id2) {
// Handle exceptions of Type2
} catch(Type3 id3) {
// Handle exceptions of Type3
}
// etc...
```

The handlers must appear directly after the **try** block. If an exception is thrown, the exception-handling mechanism goes hunting for the first handler with an argument that matches the type of the exception. Then it enters that catch clause, and the exception is considered handled. Note that within the try block, a number of different method calls might generate the same exception, but you need only one handler.

7.3 Creating Your Own Exceptions

New exceptions can be created to deal with errors not foreseen by the standard Java exceptions. To create your own exception class, you must inherit from an existing exception class, preferably one that is close in meaning to your new exception (although this is often not possible).

```
class SimpleException extends Exception {}

public class InheritingExceptions {
  public void f() throws SimpleException {
    System.out.println("Throw SimpleException from f()");
    throw new SimpleException();
  }
  public static void main(String[] args) {
```

```
    InheritingExceptions sed = new InheritingExceptions();
    try {
      sed.f();
    } catch(SimpleException e) {
      System.out.println("Caught it!");
    }
  }
}
```

You may want to send error output to the standard error stream by writing to **System.err**. This is usually a better place to send error information than **System.out**, which may be redirected. If you send output to **System.err**, it will not be redirected along with **System.out** so the user is more likely to notice it. You can also create an exception class that has a constructor with a String argument:

```
package exceptions;

//: exceptions/FullConstructors.java

class MyException extends Exception {
        public MyException() {}
        public MyException(String msg) { super(msg); }
}

public class FullConstructors {
        public static void f() throws MyException {
                System.out.println("Throwing MyException from f()");
                throw new MyException();
        }
        public static void g() throws MyException {
                System.out.println("Throwing MyException from g()");
                throw new MyException("Originated in g()");
        }
        public static void main(String[] args) {
                try {
                        f();
                } catch(MyException e) {
                        e.printStackTrace(System.out);
                }
                try {
                        g();
                } catch(MyException e) {
                        e.printStackTrace(System.out);
                }
        }
} /* Output:
```

```
Throwing MyException from f()
MyException
        at FullConstructors.f(FullConstructors.java:11)
        at FullConstructors.main(FullConstructors.java:19)
Throwing MyException from g()
MyException: Originated in g()
        at FullConstructors.g(FullConstructors.java:15)
        at FullConstructors.main(FullConstructors.java:24)
 *///:~
```

7.4 Exceptions and Logging

You may also want to log the output using the **java.util.logging** facility. One option is to build all the logging infrastructure in the exception itself (see **LoggingExceptions.java**); but it is more common that you will be catching and logging someone else 's exception, so you must generate the log message in the exception handler (see **LoggingExceptions2.java**).

Chapter 8

Miscellaneous

8.1 Primitive types, Wrapper types and Auto-boxing

The "primitive types" in Java are somehow managed with a "special treatment". The reason for the special treatment is that to create an object with **new** (especially a small, simple variable) is not very efficient, because new places objects on the heap. For these types Java falls back on the approach taken by C and C++. That is, instead of creating the variable by using new, an "automatic" variable is created that is NOT a reference. The variable holds the value directly, and it is placed on the stack, so it is much more efficient.

Primitive Type	Size	Minimum	Maximum	Wrapper Type
boolean				Boolean
char	16 bits	Unicode 0	Unicode 2^{16}-1	Character
byte	8 bits	-128	+127	Byte
short	16 bits	-2^{15}	$+2^{15}$-1	Short
int	32 bits	-2^{31}	+231-1	Integer
long	64 bits	-2^{63}	$+2^{63}$-1	Long
float	32 bits	IEEE754	IEEE754	Float
double	64 bits	IEEE754	IEEE754	Double
void				Void

The "wrapper" classes for the primitive data types allow you to make a non-primitive object on the heap to represent that primitive type. For example:

```
char c = 'x';
Character ch = new Character(c);
```

Or you could also use:

```
Character ch = new Character('x');
```

Java SE5 autoboxing will automatically convert from a primitive to a wrapper type:

```
Character ch = 'x';
```

and back:

```
char c = ch;
```

8.2 The static keyword

Ordinarily, when you create a class you are describing how objects of that class look and how they will behave. You don not actually get an object until you create one using new, and at that point storage is allocated and methods become available.
There are two situations in which this approach is not sufficient. One is if you want to have only a single piece of storage for a particular field, regardless of how many objects of that class are created, or even if no objects are created.
The other is if you need a method that is not associated with any particular object of this class. That is, you need a method that you can call even if no objects are created.

You can achieve both of these effects with the **static** keyword. When you say something is **static**, it means that particular field or method is not tied to any particular object instance of that class. So even if you have never created an object of that class you can call a **static** method or access a **static** field. With ordinary, non-static fields and methods, you must create an object and use that object to access the field or method, since non-static fields and methods must know the particular object they are working with.

For example, the following produces a static field and initializes it:

```
class StaticTest {
static int i = 47;
}
```

Now even if you make two **StaticTest** objects, there will still be only one piece of storage for **StaticTest.i**. Both objects will share the same **i**. Consider:

```
StaticTest st1 = new StaticTest();
StaticTest st2 = new StaticTest();
```

At this point, both **st1.i** and **st2.i** have the same value of 47 since they refer to the same piece of memory.

There are two ways to refer to a static variable. As the preceding example indicates, you can name it via an object, by saying, for example, **st2.i**. You can

also refer to it directly through its class name, something you cannot do with a non-static member.

```
StaticTest.i++;
```

The ++ operator adds one to the variable. At this point, both st1.i and st2.i will have the value 48.