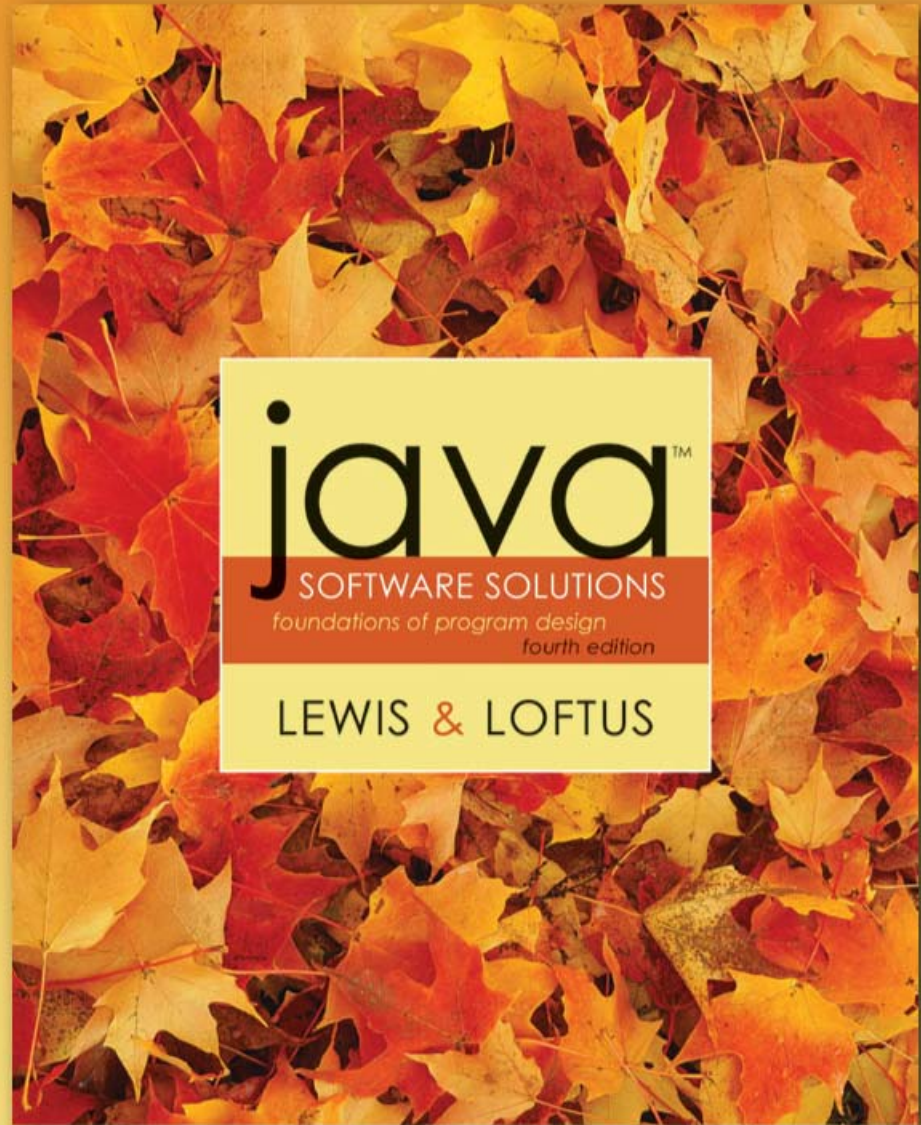


Lecture 13

Repetitorium



Identifiers

- ***Identifiers*** are the words a programmer uses in a program
- An identifier can be made up of letters, digits, the underscore character (`_`), and the dollar sign
- Identifiers cannot begin with a digit
- Java is ***case sensitive*** - `Total`, `total`, and `TOTAL` are different identifiers
- By convention, programmers use different case styles for different types of identifiers, such as
 - ***title case*** for class names - `Lincoln`
 - ***upper case*** for constants - `MAXIMUM`

Identifiers

- Sometimes we choose identifiers ourselves when writing a program (such as `Lincoln`)
- Sometimes we are using another programmer's code, so we use the identifiers that he or she chose (such as `println`)
- Often we use special identifiers called *reserved words* that already have a predefined meaning in the language
- A reserved word cannot be used in any other way

Objects

- **An object has:**
 - *state* - descriptive characteristics
 - *behaviors* - what it can do (or what can be done to it)
- **The state of a bank account includes its account number and its current balance**
- **The behaviors associated with a bank account include the ability to make deposits and withdrawals**
- **Note that the behavior of an object might change its state**

Classes

- An object is defined by a *class*
- A class is the blueprint of an object
- The class uses methods to define the behaviors of the object
- The class that contains the main method of a Java program represents the entire program
- A class represents a concept, and an object represents the embodiment of that concept
- Multiple objects can be created from the same class

Primitive Data

- There are eight primitive data types in Java
- Four of them represent integers:
 - `byte`, `short`, `int`, `long`
- Two of them represent floating point numbers:
 - `float`, `double`
- One of them represents characters:
 - `char`
- And one of them represents boolean values:
 - `boolean`

Expressions

- An *expression* is a combination of one or more operators and operands
- *Arithmetic expressions* compute numeric results and make use of the arithmetic operators:

Addition	+
Subtraction	-
Multiplication	*
Division	/
Remainder	%

- If either or both operands used by an arithmetic operator are floating point, then the result is a floating point

Operator Precedence

- Operators can be combined into complex expressions

```
result = total + count / max - offset;
```

- Operators have a well-defined precedence which determines the order in which they are evaluated
- Multiplication, division, and remainder are evaluated prior to addition, subtraction, and string concatenation
- Arithmetic operators with the same precedence are evaluated from left to right, but parentheses can be used to force the evaluation order

Assignment Operators

- There are many assignment operators in Java, including the following:

<u>Operator</u>	<u>Example</u>	<u>Equivalent To</u>
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y



Interactive Programs

- Programs generally need input on which to operate
- The `Scanner` class provides convenient methods for reading input values of various types
- A `Scanner` object can be set up to read input from various sources, including the user typing values on the keyboard
- Keyboard input is represented by the `System.in` object

Creating Objects

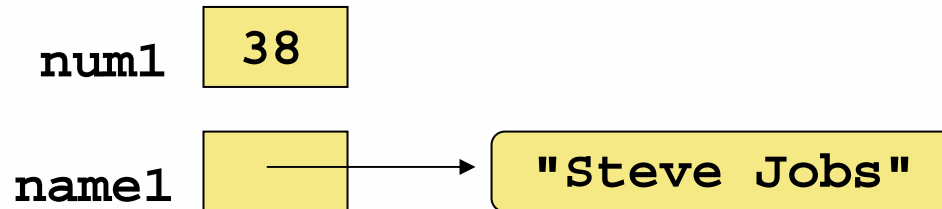
- A variable holds either a primitive type or a *reference* to an object
- A class name can be used as a type to declare an *object reference variable*

```
String title;
```

- No object is created with this declaration
- An object reference variable holds the address of an object
- The object itself must be created separately

References

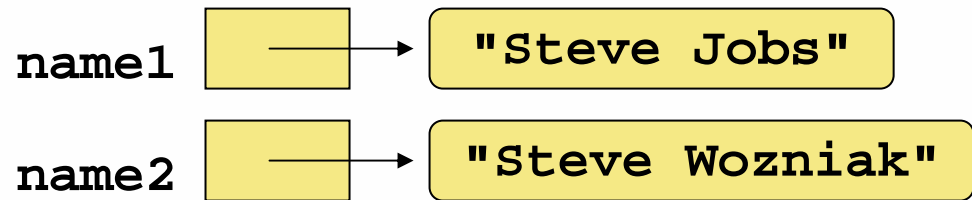
- Note that a primitive variable contains the value itself, but an object variable contains the address of the object
- An object reference can be thought of as a pointer to the location of the object
- Rather than dealing with arbitrary addresses, we often depict a reference graphically



Reference Assignment

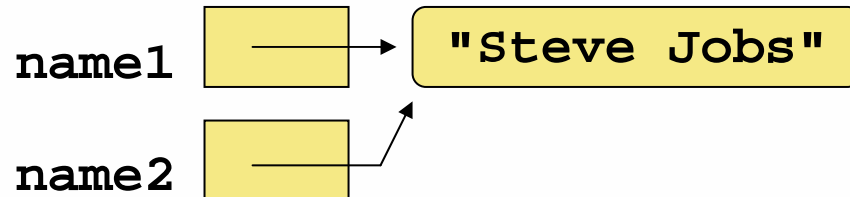
- For object references, assignment copies the address:

Before:



`name2 = name1;`

After:



String Indexes

- It is occasionally helpful to refer to a particular character within a string
- This can be done by specifying the character's numeric *index*
- The indexes begin at zero in each string
- In the string "Hello", the character 'H' is at index 0 and the 'o' is at index 4
- See [StringMutation.java](#) (page 120)

Wrapper Classes

- The `java.lang` package contains *wrapper classes* that correspond to each primitive type:

<u>Primitive Type</u>	<u>Wrapper Class</u>
<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>char</code>	<code>Character</code>
<code>boolean</code>	<code>Boolean</code>
<code>void</code>	<code>Void</code>

Autoboxing

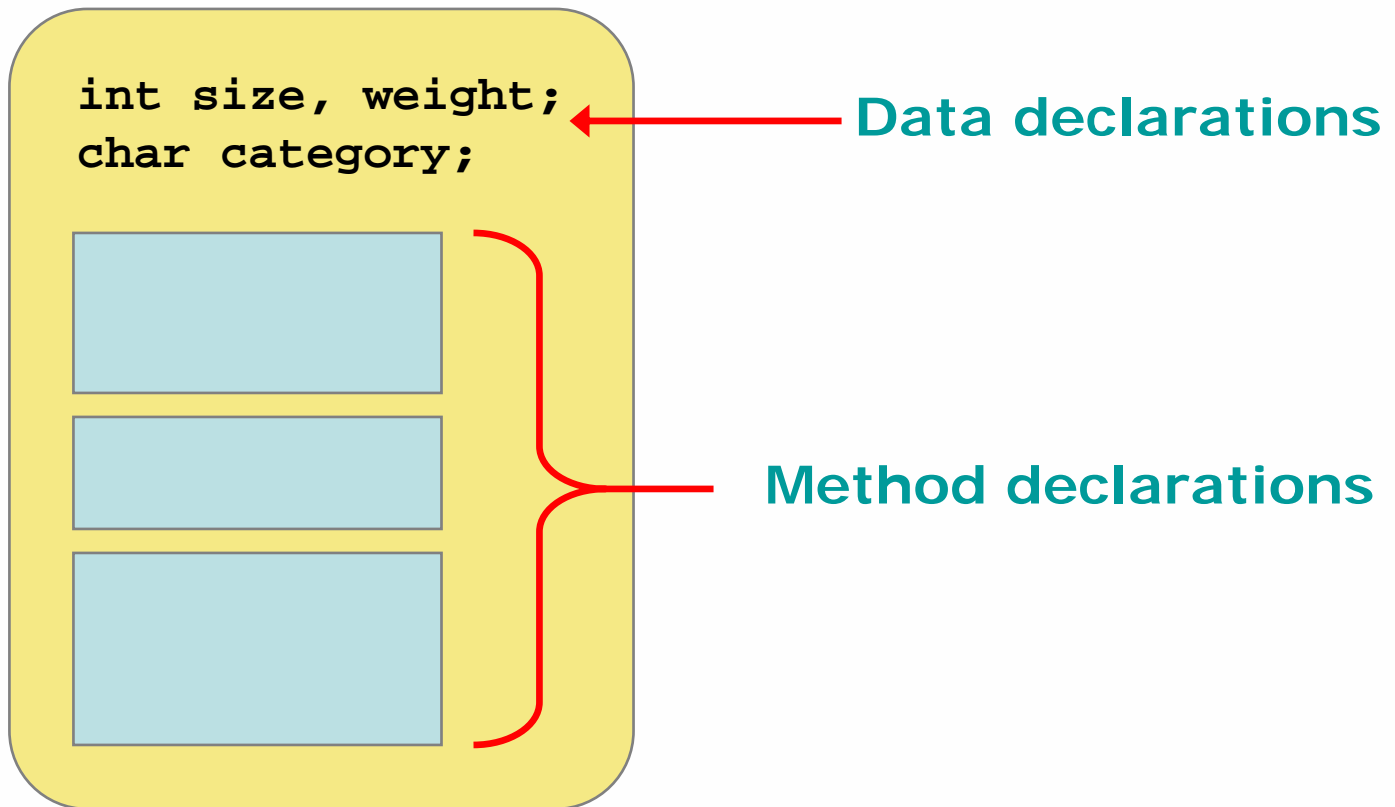
- ***Autoboxing*** is the automatic conversion of a primitive value to a corresponding wrapper object:

```
Integer obj;  
int num = 42;  
obj = num;
```

- The assignment creates the appropriate `Integer` object
- The reverse conversion (called *unboxing*) also occurs automatically as needed

Classes

- A class can contain data declarations and method declarations



Classes

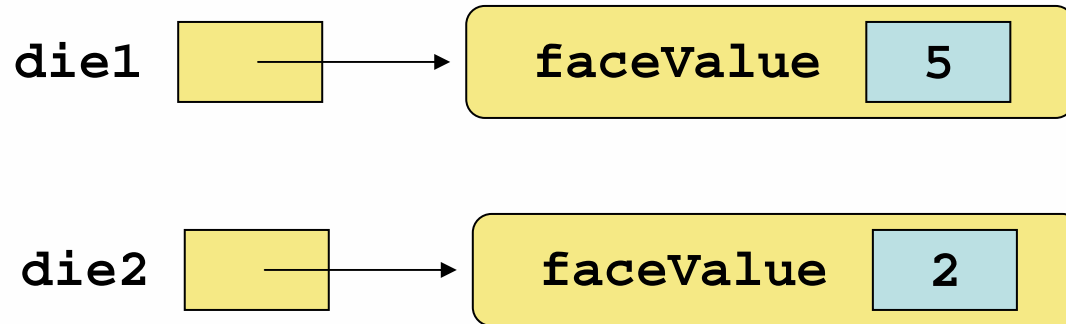
- We'll want to design the `Die` class with other data and methods to make it a versatile and reusable resource
- Any given program will not necessarily use all aspects of a given class
- See [RollingDice.java](#) (page 157)
- See [Die.java](#) (page 158)

Instance Data

- The `faceValue` variable in the `Die` class is called *instance data* because each instance (object) that is created has its own version of it
- A class declares the type of the data, but it does not reserve any memory space for it
- Every time a `Die` object is created, a new `faceValue` variable is created as well
- The objects of a class share the method definitions, but each object has its own data space
- That's the only way two objects can have different states

Instance Data

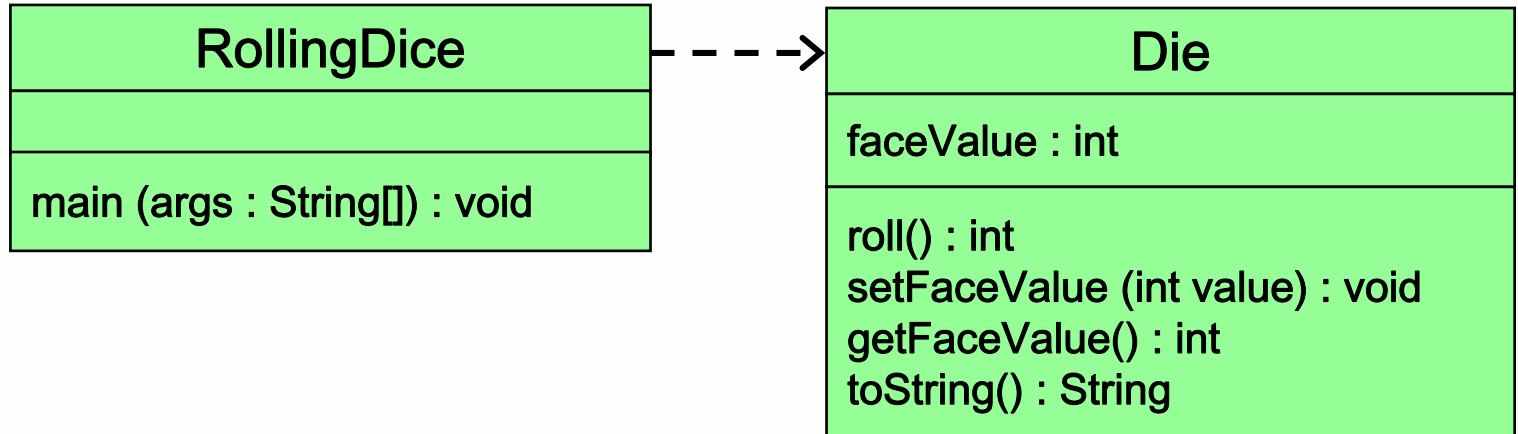
- We can depict the two `Die` objects from the `RollingDice` program as follows:



Each object maintains its own `faceValue` variable, and thus its own state

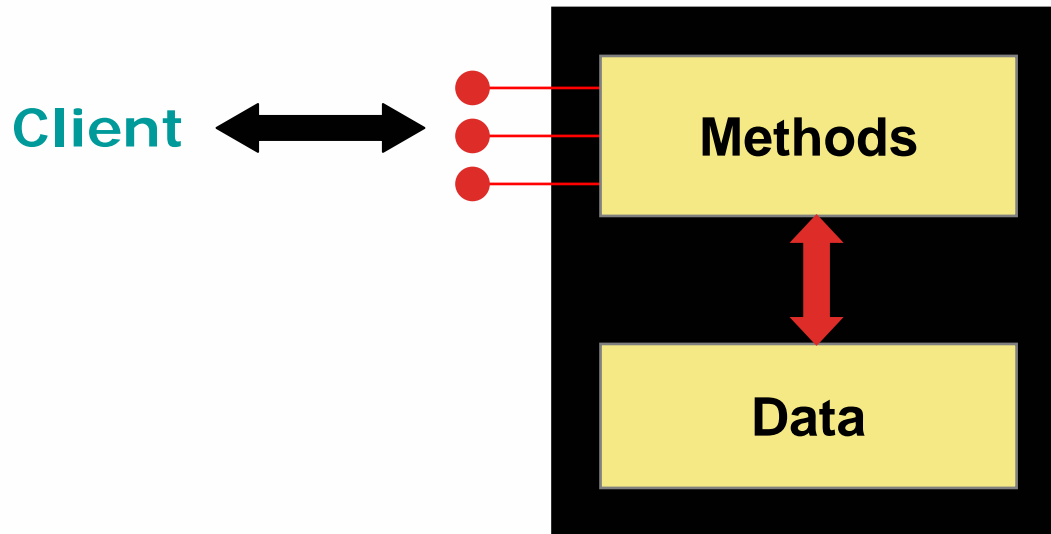
UML Class Diagrams

- A UML class diagram for the RollingDice program:



Encapsulation

- An encapsulated object can be thought of as a *black box* -- its inner workings are hidden from the client
- The client invokes the interface methods of the object, which manages the instance data



Visibility Modifiers

	<code>public</code>	<code>private</code>
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

Method Header

- A method declaration begins with a *method header*

```
char calc (int num1, int num2, String message)
```

↑
return
type

↑
method
name

parameter list

The parameter list specifies the type and name of each parameter

The name of a parameter in the method declaration is called a *formal parameter*

Method Body

- The method header is followed by the *method body*

```
char calc (int num1, int num2, String message)
{
    int sum = num1 + num2;
    char result = message.charAt (sum);

    return result;
}
```

The return expression
must be consistent with
the return type

sum and result
are local data

They are created
each time the
method is called, and
are destroyed when
it finishes executing

The if Statement

- The *if statement* has the following syntax:

`if` is a Java reserved word

The *condition* must be a boolean expression. It must evaluate to either true or false.

`if (condition)
 statement ;`

If the *condition* is true, the *statement* is executed.
If it is false, the *statement* is skipped.

Logical Operators

- Boolean expressions can also use the following *logical operators*:

!	Logical NOT
&&	Logical AND
	Logical OR

- They all take boolean operands and produce boolean results
- Logical NOT is a unary operator (it operates on one operand)
- Logical AND and logical OR are binary operators (each operates on two operands)

The if-else Statement

- An *else clause* can be added to an `if` statement to make an *if-else statement*

```
if ( condition )  
    statement1;  
else  
    statement2;
```

- If the *condition* is true, *statement1* is executed; if the condition is false, *statement2* is executed
- One or the other will be executed, but not both
- See [Wages.java](#) (page 211)

Indentation Revisited

- Remember that indentation is for the human reader, and is ignored by the computer

```
if (total > MAX)
    System.out.println ("Error!!");
    errorCount++;
```

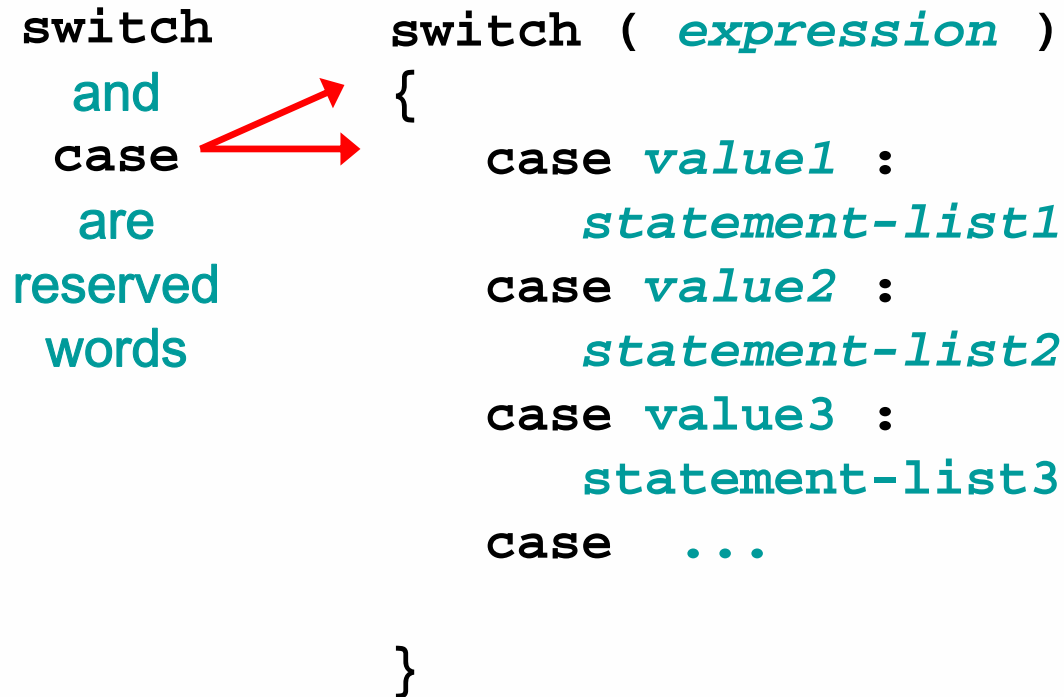


Despite what is implied by the indentation, the increment will occur whether the condition is true or not

The switch Statement

- The general syntax of a switch statement is:

switch
and
case
are
reserved
words



```
switch ( expression )  
{  
    case value1 :  
        statement-list1  
    case value2 :  
        statement-list2  
    case value3 :  
        statement-list3  
    case ...  
}
```

If *expression*
matches *value2*,
control jumps
to here

The while Statement

- A *while statement* has the following syntax:

```
while ( condition )  
    statement;
```

- If the *condition* is true, the *statement* is executed
- Then the condition is evaluated again, and if it is still true, the statement is executed again
- The statement is executed repeatedly until the condition becomes false

Nested Loops

- How many times will the string "Here" be printed?

```
count1 = 1;
while (count1 <= 10)
{
    count2 = 1;
    while (count2 <= 20)
    {
        System.out.println ("Here");
        count2++;
    }
    count1++;
}
```

10 * 20 = 200

The do Statement

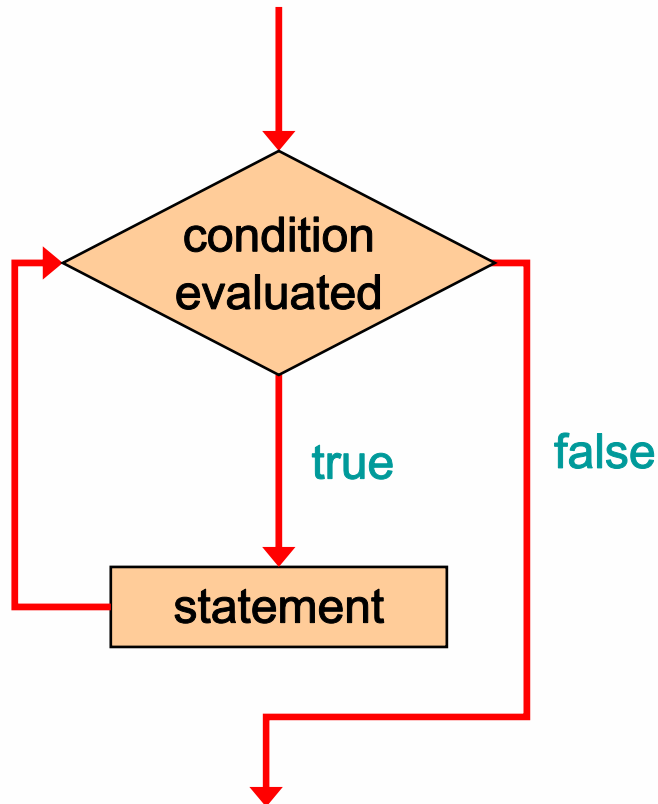
- A *do statement* has the following syntax:

```
do
{
    statement;
}
while ( condition )
```

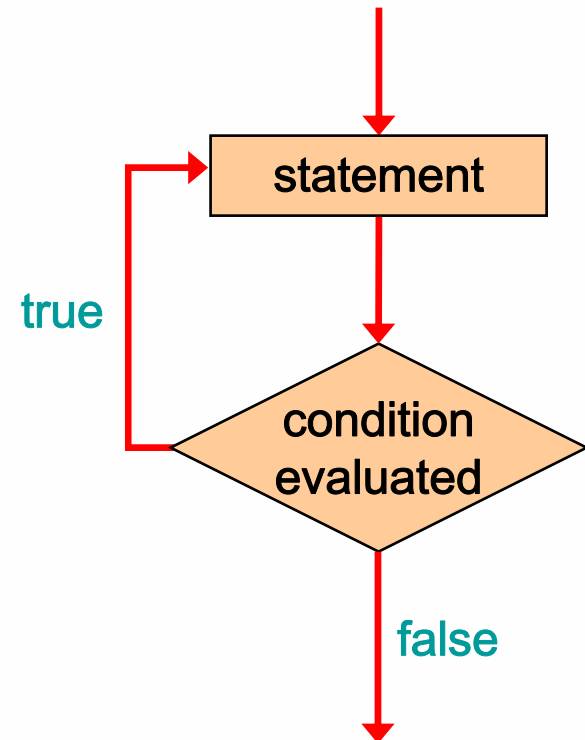
- The *statement* is executed once initially, and then the *condition* is evaluated
- The statement is executed repeatedly until the condition becomes false

Comparing while and do

The while Loop



The do Loop



The for Statement

- A *for statement* has the following syntax:

The *initialization*
is executed once
before the loop begins



The *statement* is
executed until the
condition becomes false



```
for ( initialization ; condition ; increment )  
    statement;
```



The *increment* portion is executed at
the end of each iteration

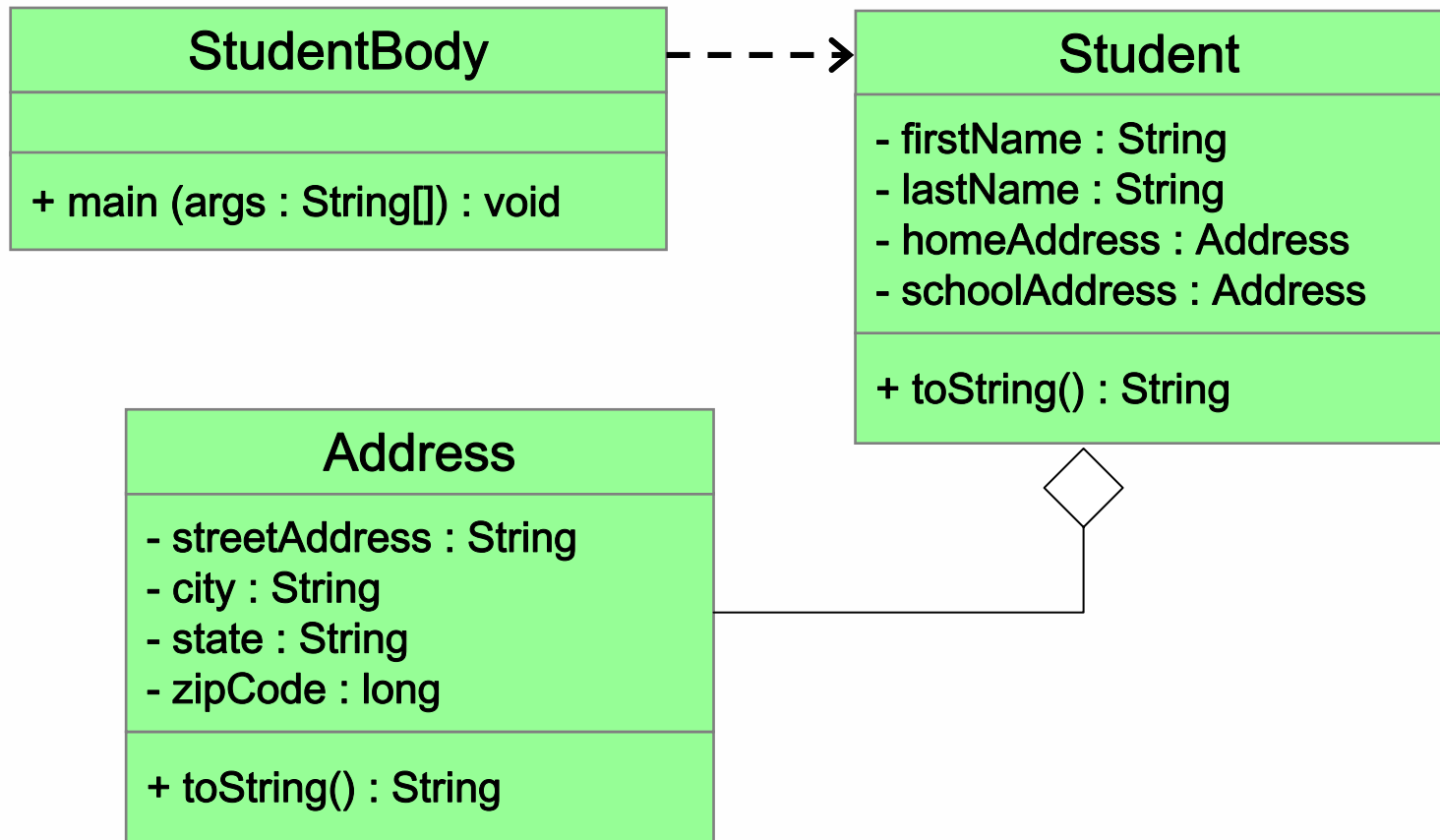
Static Class Members

- Static methods and static variables often work together
- The following example keeps track of how many `Slogan` objects have been created using a static variable, and makes that information available using a static method
- See [SloganCounter.java](#) (page 294)
- See [Slogan.java](#) (page 295)

Class Relationships

- **Classes in a software system can have various types of relationships to each other**
- **Three of the most common relationships:**
 - **Dependency: A *uses* B**
 - **Aggregation: A *has-a* B**
 - **Inheritance: A *is-a* B**
- **Let's discuss dependency and aggregation further**
- **Inheritance is discussed in detail in Chapter 8**

Aggregation in UML



The this reference

- The `this` reference can be used to distinguish the instance variables of a class from corresponding method parameters with the same names
- The constructor of the `Account` class (from Chapter 4) could have been written as follows:

```
public Account (String name, long acctNumber,  
                double balance)  
{  
    this.name = name;  
    this.acctNumber = acctNumber;  
    this.balance = balance;  
}
```


Interfaces

- A Java *interface* is a collection of abstract methods and constants
- An *abstract method* is a method header without a method body
- An abstract method can be declared using the modifier `abstract`, but because all methods in an interface are abstract, usually it is left off
- An interface is used to establish a set of methods that a class will implement

Interfaces

- A class that implements an interface can implement other methods as well
- See [Complexity.java](#) (page 310)
- See [Question.java](#) (page 311)
- See [MiniQuiz.java](#) (page 313)
- In addition to (or instead of) abstract methods, an interface can contain constants
- When a class implements an interface, it gains access to all its constants

Objects as Parameters

- Another important issue related to method design involves parameter passing
- Parameters in a Java method are *passed by value*
- A copy of the actual parameter (the value passed in) is stored into the formal parameter (in the method header)
- Therefore passing parameters is similar to an assignment statement
- When an object is passed to a method, the actual parameter and the formal parameter become aliases of each other

Passing Objects to Methods

- What a method does with a parameter may or may not have a permanent effect (outside the method)
- See [ParameterTester.java](#) (page 327)
- See [ParameterModifier.java](#) (page 329)
- See [Num.java](#) (page 330)
- Note the difference between changing the internal state of an object versus changing which object a reference points to

Method Overloading

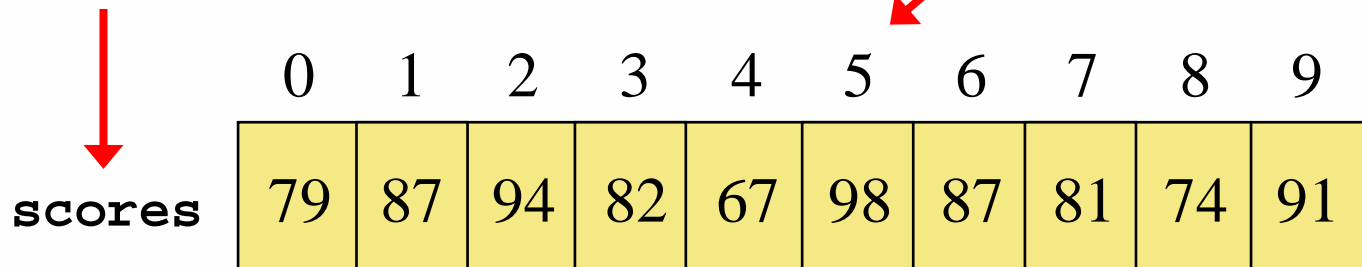
- ***Method overloading*** is the process of giving a single method name multiple definitions
- If a method is overloaded, the method name is not sufficient to determine which method is being called
- The ***signature*** of each overloaded method must be unique
- The signature includes the number, type, and order of the parameters

Arrays

- An *array* is an ordered list of values

The entire array
has a single name

Each value has a numeric *index*



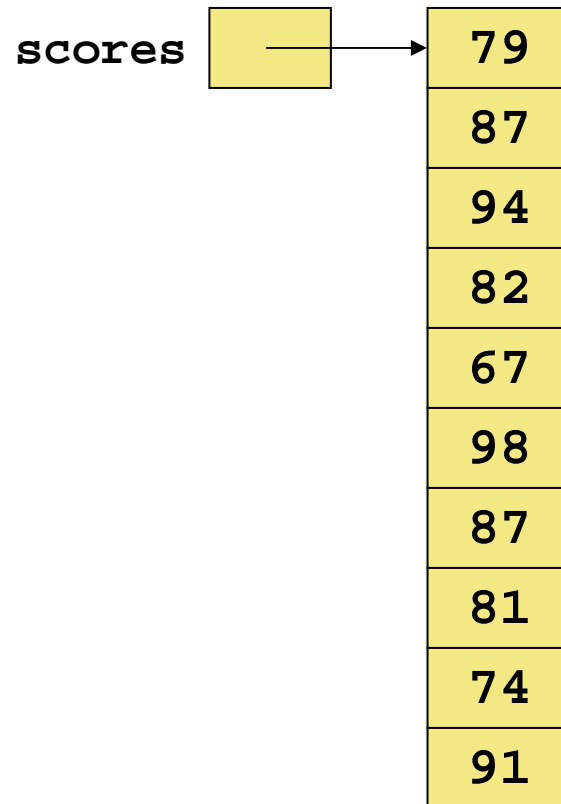
	0	1	2	3	4	5	6	7	8	9
scores	79	87	94	82	67	98	87	81	74	91

An array of size N is indexed from zero to N-1

This array holds 10 values that are indexed from 0 to 9

Arrays

- Another way to depict the `scores` array:

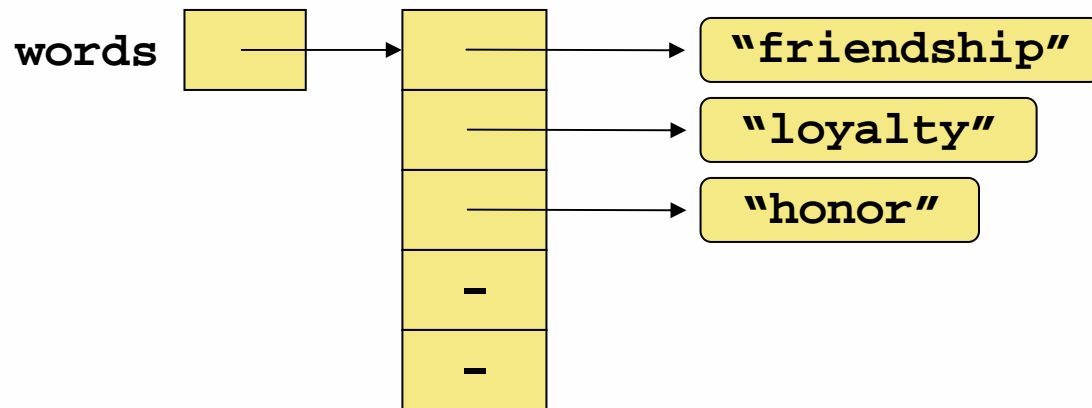


Arrays as Parameters

- **An entire array can be passed as a parameter to a method**
- **Like any other object, the reference to the array is passed, making the formal and actual parameters aliases of each other**
- **Therefore, changing an array element within the method changes the original**
- **An individual array element can be passed to a method as well, in which case the type of the formal parameter is the same as the element type**

Arrays of Objects

- After some `String` objects are created and stored in the array:

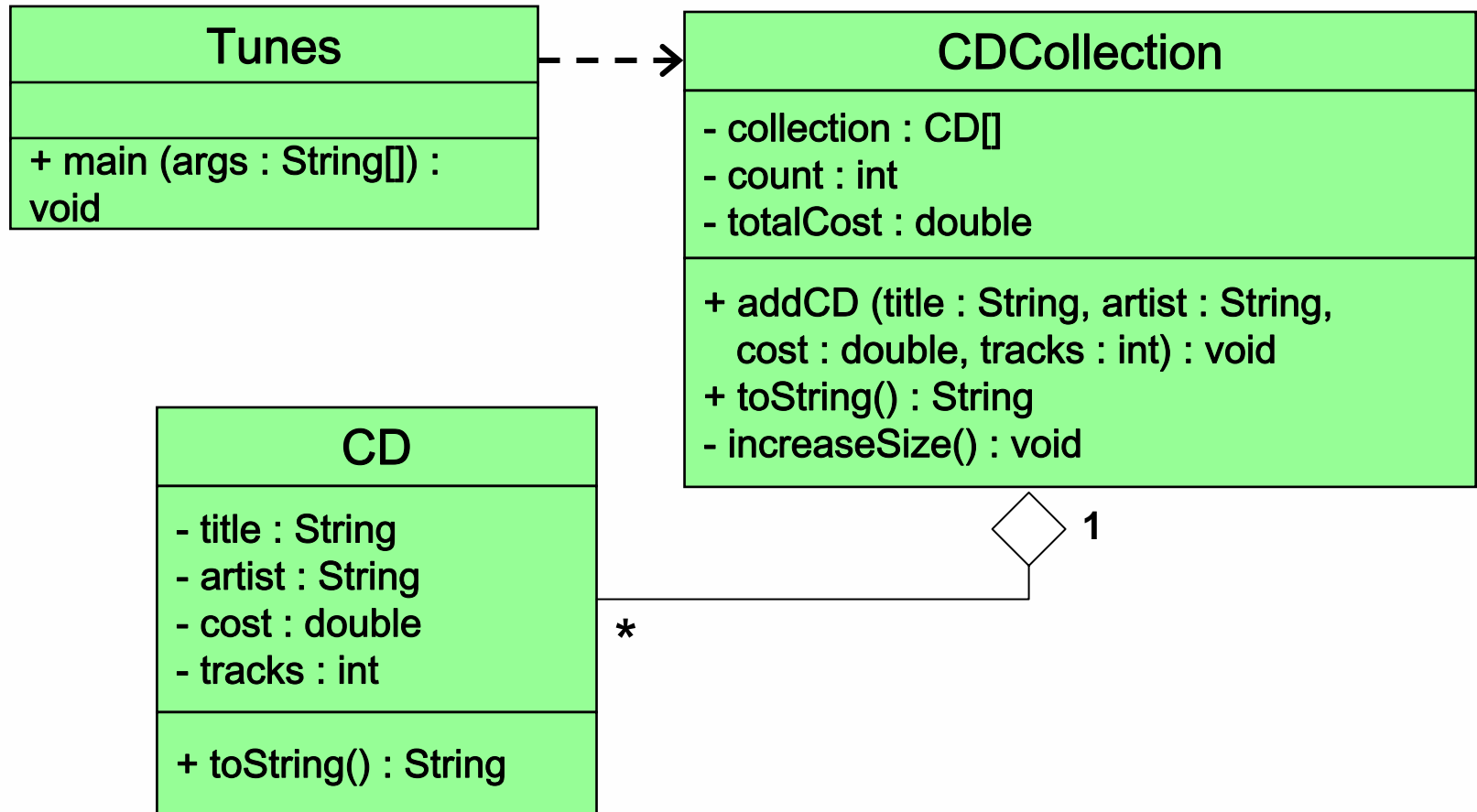


Arrays of Objects

- The following example creates an array of `Grade` objects, each with a string representation and a numeric lower bound
- See [GradeRange.java](#) (page 384)
- See [Grade.java](#) (page 385)
- Now let's look at an example that manages a collection of `CD` objects
- See [Tunes.java](#) (page 387)
- See [CDCollection.java](#) (page 388)
- See [CD.java](#) (page 391)

Arrays of Objects

- A UML diagram for the Tunes program:



Command-Line Arguments

- The signature of the `main` method indicates that it takes an array of `String` objects as a parameter
- These values come from *command-line arguments* that are provided when the interpreter is invoked
- For example, the following invocation of the interpreter passes three `String` objects into `main`:

```
> java StateEval pennsylvania texas arizona
```
- These strings are stored at indexes 0-2 of the array parameter of the `main` method
- See [NameTag.java](#) (page 393)

The ArrayList Class

- An `ArrayList` stores references to the `Object` class, which allows it to store any kind of object
- See [Beatles.java](#) (page 405)
- We can also define an `ArrayList` object to accept a particular type of object
- The following declaration creates an `ArrayList` object that only stores `Family` objects

```
ArrayList<Family> reunion = new ArrayList<Family>
```

- This is an example of *generics*, which are discussed further in Chapter 12

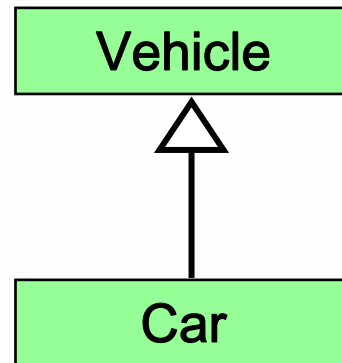


Inheritance

- ***Inheritance*** allows a software developer to derive a new class from an existing one
- The existing class is called the ***parent class***, or ***superclass***, or ***base class***
- The derived class is called the ***child class*** or ***subclass***
- As the name implies, the child inherits characteristics of the parent
- That is, the child class inherits the methods and data defined by the parent class

Inheritance

- Inheritance relationships are shown in a UML class diagram using a solid arrow with an unfilled triangular arrowhead pointing to the parent class



- Proper inheritance creates an *is-a* relationship, meaning the child *is a* more specific version of the parent



Inheritance

- A programmer can tailor a derived class as needed by adding new variables or methods, or by modifying the inherited ones
- *Software reuse* is a fundamental benefit of inheritance
- By using existing software components to create new ones, we capitalize on all the effort that went into the design, implementation, and testing of the existing software

The super Reference

- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- The `super` reference can be used to refer to the parent class, and often is used to invoke the parent's constructor
- See [Words2.java](#) (page 445)
- See [Book2.java](#) (page 446)
- See [Dictionary2.java](#) (page 447)

The super Reference

- A child's constructor is responsible for calling the parent's constructor
- The first line of a child's constructor should use the `super` reference to call the parent's constructor
- The `super` reference can also be used to reference other variables and methods defined in the parent's class

Overriding Methods

- A child class can *override* the definition of an inherited method in favor of its own
- The new method must have the same signature and return type as the parent's method, but can have a different body
- The type of the object executing the method determines which version of the method is invoked
- See [Messages.java](#) (page 450)
- See [Thought.java](#) (page 451)
- See [Advice.java](#) (page 452)

Overriding

- A method in the parent class can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code



Overloading vs. Overriding

- **Overloading deals with multiple methods with the same name in the same class, but with different signatures**
- **Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature**
- **Overloading lets you define a similar operation in different ways for different parameters**
- **Overriding lets you define a similar operation in different ways for different object types**

Abstract Classes

- An *abstract class* is a placeholder in a class hierarchy that represents a generic concept
- An abstract class cannot be instantiated
- We use the modifier `abstract` on the class header to declare a class as abstract:

```
public abstract class Product
{
    // contents
}
```



Visibility Revisited

- **It's important to understand one subtle issue related to inheritance and visibility**
- **All variables and methods of a parent class, except private methods, are inherited by its children**
- **As we've mentioned, private members cannot be referenced by name in the child class**
- **However, private members can be referenced indirectly**

Binding

- Consider the following method invocation:

```
obj.doIt();
```

- At some point, this invocation is *bound* to the definition of the method that it invokes
- If this binding occurred at compile time, then that line of code would call the same method every time
- However, Java defers method binding until run time -- this is called *dynamic binding* or *late binding*
- Late binding provides flexibility in program design



Polymorphism

- The term *polymorphism* literally means "having many forms"
- A *polymorphic reference* is a variable that can refer to different types of objects at different points in time
- The method invoked through a polymorphic reference can change from one invocation to the next
- All object references in Java are potentially polymorphic

Polymorphism

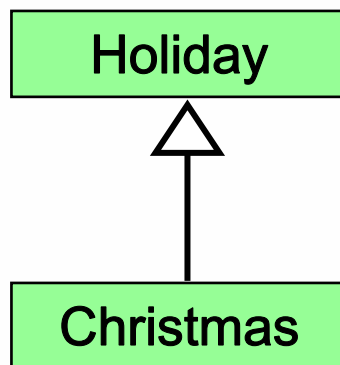
- Suppose we create the following reference variable:

```
Occupation job;
```

- Java allows this reference to point to an `Occupation` object, or to any object of any compatible type
- This compatibility can be established using inheritance or using interfaces
- Careful use of polymorphic references can lead to elegant, robust software designs

References and Inheritance

- An object reference can refer to an object of its class, or to an object of any class related to it by inheritance
- For example, if the `Holiday` class is used to derive a class called `Christmas`, then a `Holiday` reference could be used to point to a `Christmas` object



```
Holiday day;  
day = new Christmas();
```

Polymorphism via Inheritance

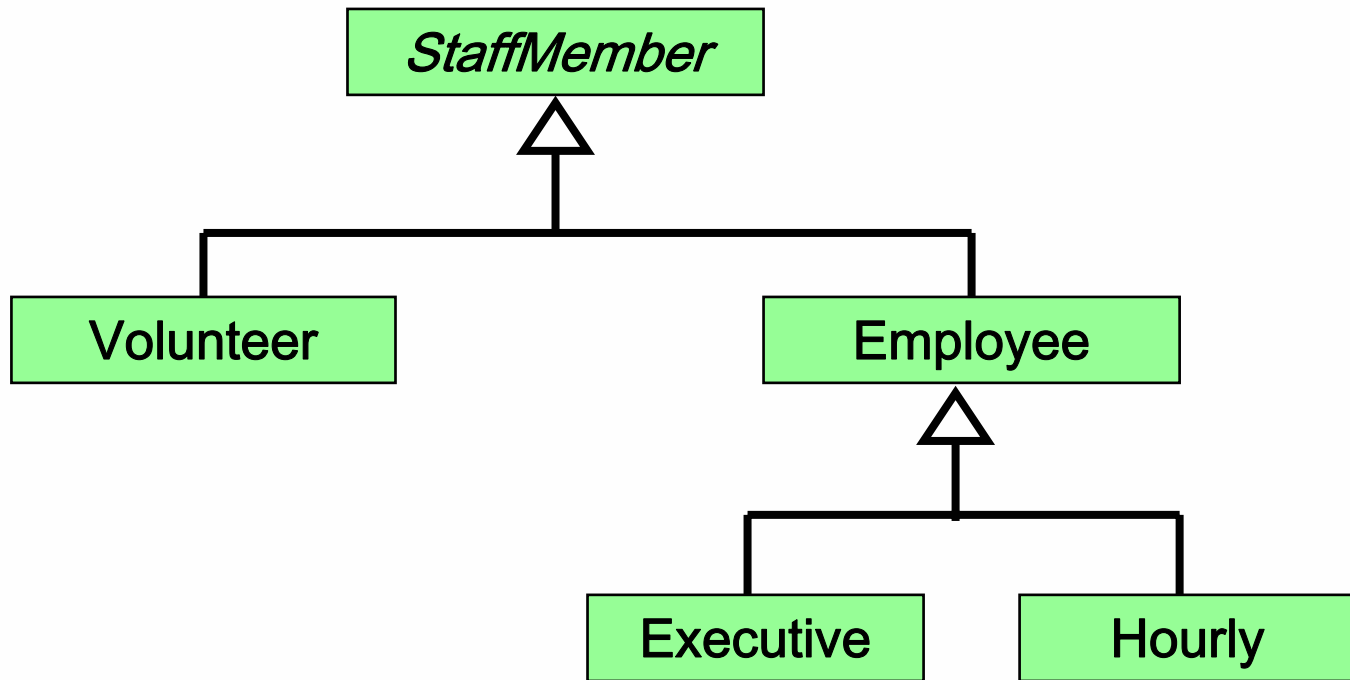
- It is the type of the object being referenced, not the reference type, that determines which method is invoked
- Suppose the `Holiday` class has a method called `celebrate`, and the `Christmas` class overrides it
- Now consider the following invocation:

```
day.celebrate( );
```

- If `day` refers to a `Holiday` object, it invokes the `Holiday` version of `celebrate`; if it refers to a `Christmas` object, it invokes the `Christmas` version

Polymorphism via Inheritance

- Consider the following class hierarchy:



Polymorphism via Inheritance

- Now let's look at an example that pays a set of diverse employees using a polymorphic method
- See [Firm.java](#) (page 486)
- See [Staff.java](#) (page 487)
- See [StaffMember.java](#) (page 489)
- See [Volunteer.java](#) (page 491)
- See [Employee.java](#) (page 492)
- See [Executive.java](#) (page 493)
- See [Hourly.java](#) (page 494)

Polymorphism via Interfaces

- An interface name can be used as the type of an object reference variable

```
Speaker current;
```

- The `current` reference can be used to point to any object of any class that implements the `Speaker` interface
- The version of `speak` that the following line invokes depends on the type of object that `current` is referencing

```
current.speak();
```

Polymorphism via Interfaces

- Suppose two classes, `Philosopher` and `Dog`, both implement the `Speaker` interface, providing distinct versions of the `speak` method
- In the following code, the first call to `speak` invokes one version and the second invokes another:

```
Speaker guest = new Philosopher();  
guest.speak();  
guest = new Dog();  
guest.speak();
```

Selection Sort

- The sorting method doesn't "care" what it is sorting, it just needs to be able to call the `compareTo` method
- That is guaranteed by using `Comparable` as the parameter type
- Also, this way each class decides for itself what it means for one object to be less than another
- See [PhoneList.java](#) (page 500)
- See [Sorting.java](#) (page 501), specifically the `selectionSort` method
- See [Contact.java](#) (page 503)

Linear Search

- A linear search begins at one end of a list and examines each element in turn
- Eventually, either the item is found or the end of the list is encountered
- See [PhoneList2.java](#) (page 508)
- See [Searching.java](#) (page 509), specifically the `linearSearch` method

Binary Search

- A *binary search* assumes the list of items in the search pool is sorted
- It eliminates a large part of the search pool with a single comparison
- A binary search first examines the middle element of the list -- if it matches the target, the search is over
- If it doesn't, only one half of the remaining elements need be searched
- Since they are sorted, the target can only be in one half of the other

Binary Search

- The process continues by comparing the middle element of the remaining *viable candidates*
- Each comparison eliminates approximately half of the remaining data
- Eventually, the target is found or the data is exhausted
- See [PhoneList2.java](#) (page 508)
- See [Searching.java](#) (page 509), specifically the `binarySearch` method

Exceptions

- An *exception* is an object that describes an unusual or erroneous situation
- Exceptions are *thrown* by a program, and may be *caught* and *handled* by another part of the program
- A program can be separated into a normal execution flow and an *exception execution flow*
- An *error* is also represented as an object in Java, but usually represents a unrecoverable situation and should not be caught

Exception Handling

- **Java has a predefined set of exceptions and errors that can occur during execution**
- **A program can deal with an exception in one of three ways:**
 - **ignore it**
 - **handle it where it occurs**
 - **handle it in another place in the program**
- **The manner in which an exception is processed is an important design consideration**

The try Statement

- To handle an exception in a program, the line that throws the exception is executed within a *try block*
- A try block is followed by one or more *catch* clauses
- Each catch clause has an associated exception type and is called an *exception handler*
- When an exception occurs, processing continues at the first catch clause that matches the exception type
- See [ProductCodes.java](#) (page 536)

Exception Propagation

- An exception can be handled at a higher level if it is not appropriate to handle it where it occurs
- Exceptions *propagate* up through the method calling hierarchy until they are caught and handled or until they reach the level of the `main` method
- A try block that contains a call to a method in which an exception is thrown can be used to catch that exception
- See [Propagation.java](#) (page 539)
- See [ExceptionScope.java](#) (page 540)

Checked Exceptions

- An exception is either *checked* or *unchecked*
- A *checked exception* either must be caught by a method, or must be listed in the *throws clause* of any method that may throw or propagate it
- A *throws clause* is appended to the method header
- The compiler will issue an error if a checked exception is not caught or asserted in a *throws clause*

Unchecked Exceptions

- An unchecked exception does not require explicit handling, though it could be processed that way
- The only unchecked exceptions in Java are objects of type `RuntimeException` or any of its descendants
- Errors are similar to `RuntimeException` and its descendants in that:
 - Errors should not be caught
 - Errors do not require a throws clause

The throw Statement

- Exceptions are thrown using the *throw* statement
- Usually a throw statement is executed inside an if statement that evaluates a condition to see if the exception should be thrown
- See [CreatingExceptions.java](#) (page 543)
- See [OutOfRangeException.java](#) (page 544)



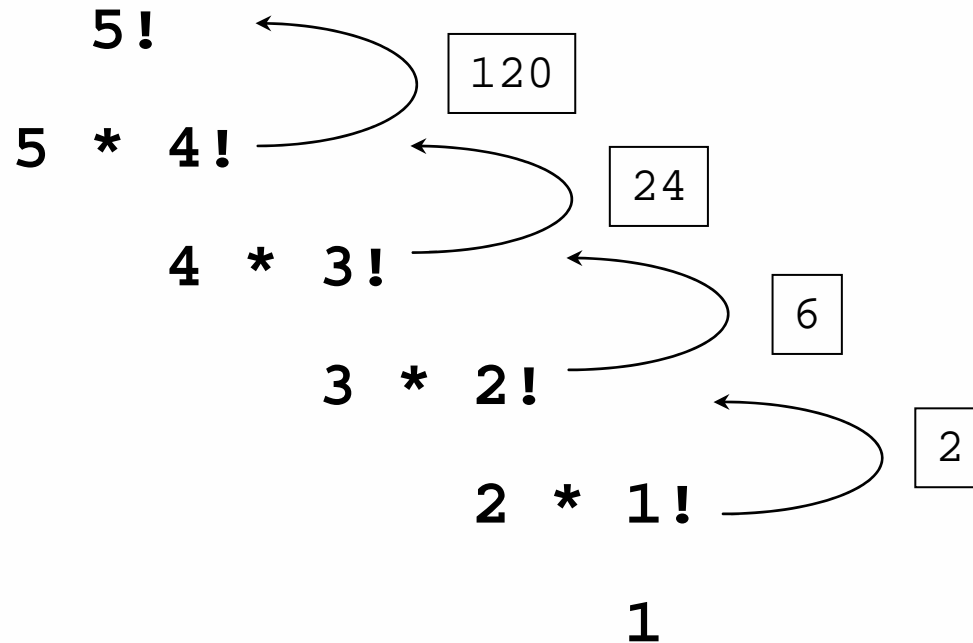
Recursive Thinking

- ***A recursive definition* is one which uses the word or concept being defined in the definition itself**
- **When defining an English word, a recursive definition is often not helpful**
- **But in other situations, a recursive definition can be an appropriate way to express a concept**
- **Before applying recursion to programming, it is best to practice thinking recursively**

Infinite Recursion

- All recursive definitions have to have a non-recursive part
- If they didn't, there would be no way to terminate the recursive path
- Such a definition would cause *infinite recursion*
- This problem is similar to an infinite loop, but the non-terminating "loop" is part of the definition itself
- The non-recursive part is often called the *base case*

Recursive Definitions



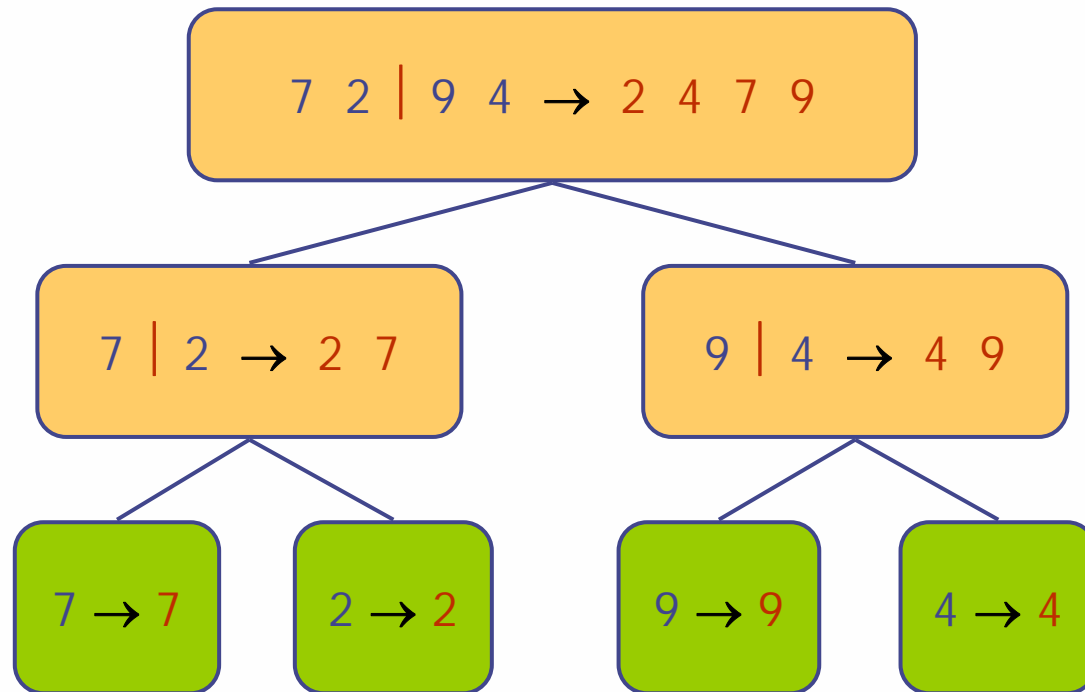
Recursive Programming

- A method in Java can invoke itself; if set up that way, it is called a *recursive method*
- The code of a recursive method must be structured to handle both the base case and the recursive case
- Each call to the method sets up a new execution environment, with new parameters and local variables
- As with any method call, when the method completes, control returns to the method that invoked it (which may be an earlier invocation of itself)

Towers of Hanoi

- An iterative solution to the Towers of Hanoi is quite complex
- A recursive solution is much shorter and more elegant
- See [SolveTowers.java](#) (page 590)
- See [TowersOfHanoi.java](#) (page 591)

Merge Sort



Divide and Conquer

- **The Merge Sort algorithm sorts an array by**
 - cutting the array in half
 - recursively sorting each half
 - merging the sorted halves
- **The recursion terminates if we are left with subarrays of length one**
- **Thus, the algorithm keeps dividing the array into smaller and smaller subarrays, sorting each half and merging them back together**
- **This algorithm carries out dramatically fewer steps than the insertion or selection sort algorithms**

Complexity of Merge Sort

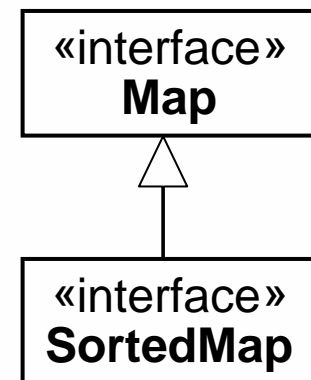
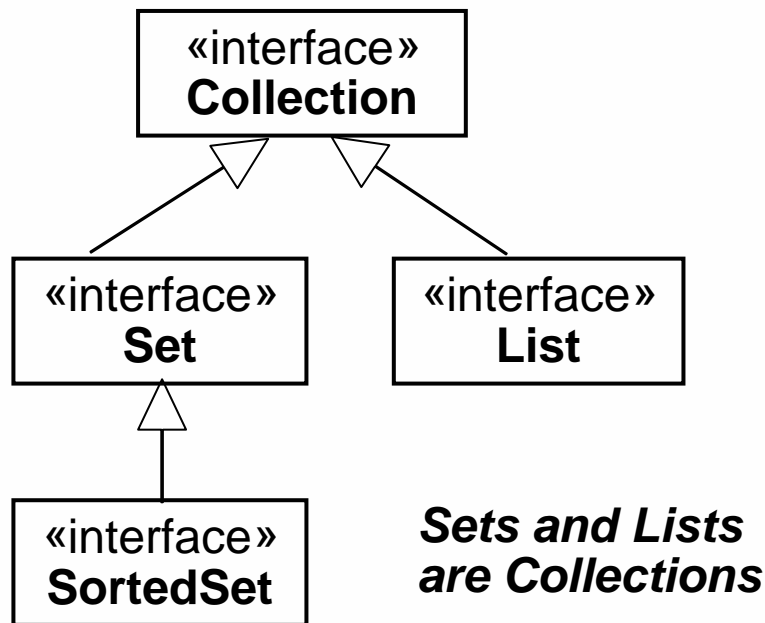
- The complexity of Merge Sort is $O(n \log n)$, where n is the size of the input
- Recall that Selection Sort and Insertion Sort have complexity $O(n^2)$
- There are further efficient, recursively formulated sorting algorithms, e.g.
 - Quick Sort
- You will learn more about searching and sorting in the lecture “Datenstrukturen und Algorithmen”

Magazine Collection

- Let's explore an example of a collection of Magazine objects, managed by the MagazineList class, which has a private inner class called MagazineNode
- Because the MagazineNode is private to MagazineList, the MagazineList methods can directly access MagazineNode data without violating encapsulation
- See [MagazineRack.java](#) (page 615)
- See [MagazineList.java](#) (page 616)
- See [Magazine.java](#) (page 618)

The Collections Framework

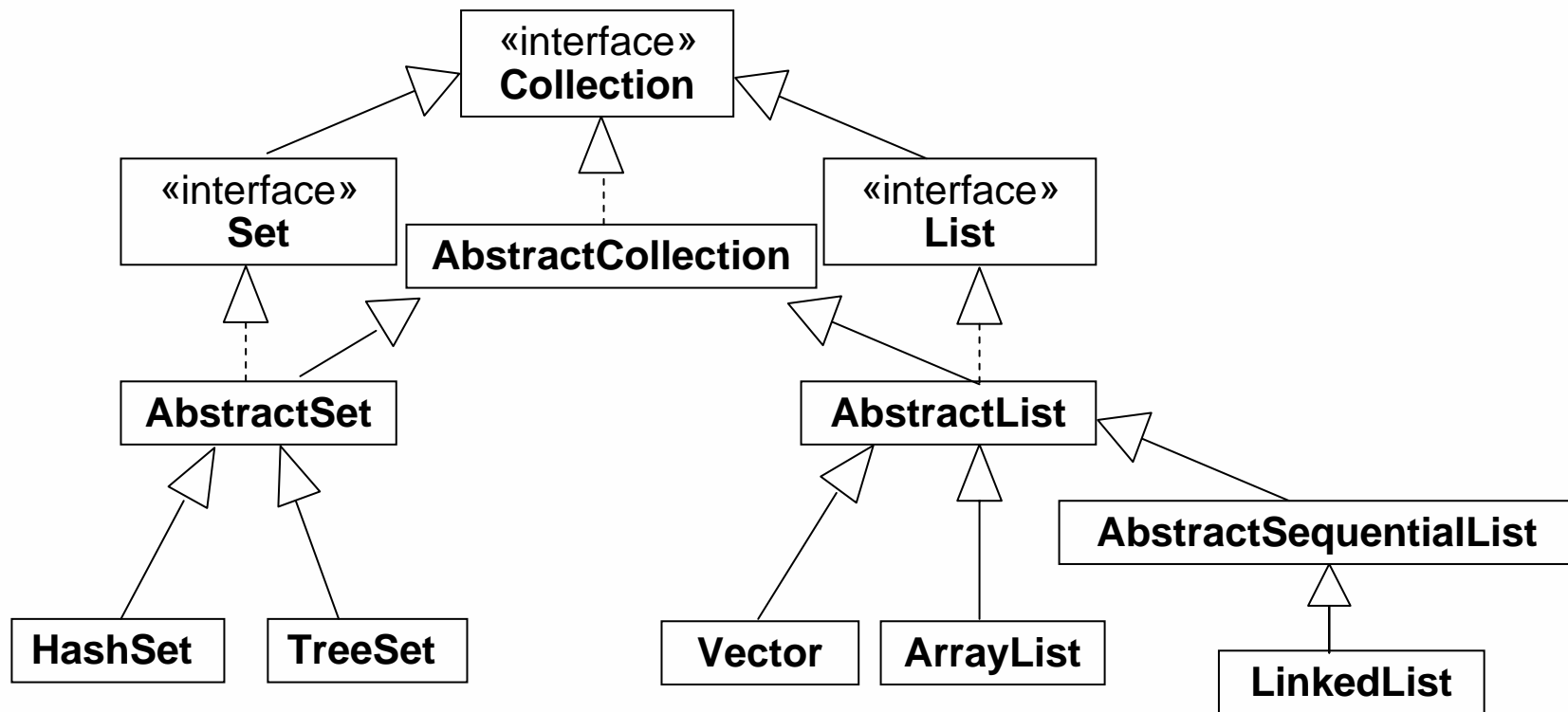
The Java Collections framework (in `java.util`) consists of *interfaces*, *implementations* and *algorithms* for manipulating collections of elements.



Maps provide mappings from keys to values

Implementations

There are at least two implementations for each interface





Some examples of generic collections

- A program that demonstrates the `LinkedList` class (see [ListTester.java](#))
- A program that demonstrates the `HashSet` class (see [SetTester.java](#))
- A program that demonstrates the `HashMap` class (see [MapTester.java](#))