Chapter 7: Object Oriented Design

7.1 Software Development Activities

Wait, how did we end up in COMP-1712?

Software development is a complex process that goes beyond just writing code. As projects grow in size and complexity, proper planning becomes essential.

There are four core development activities:

- establishing the requirements
- creating a design
- implementing the design
- testing

These activities are not strictly sequential - they overlap and interact.

1. **Establishing the Requirements**
    - Defines *what* the program must accomplish (not *how* to do it)
    - Often developed iteratively in conjunction with the client.
    - Requirements are developed iteratively because it is difficult to establish detailed, unambiguous and complete requirements.
    - Usually documented in a *functional specification*
2. **Creating a Design**
    - Specifies *how* the program will meet requirements
    - An object-oriented design defines classes, objects, and their interactions
    - Changes are easier to make during design phase
    - Includes both high-level architecture and low-level method design
3. **Implementing the Design**
    - Translates design into actual code
    - Should be least creative phase - major decisions are made during design
    - Often overemphasized by newer programmers
4. **Testing**
    - Ensures program meets requirements and constraints
    - Involves multiple test runs with various inputs
    - Requires thorough result analysis

Validating Parameters

Methods often have restrictions on parameter values that need to be validated. For example:

- `Integer.parseInt(String arg)` needs argument in integer format
- `Math.sqrt(double x)` requires a non-negative number

- How to handle invalid parameters:
    - `Integer.parseInt` throws `NumberFormatException` for non-integer strings
    - `Math.sqrt` returns `NaN` for negative numbers
        - only possible because Math.sqrt returns a double, which can use the IEEE standard NaN special case we learned about in COMP-1113

Handling Exceptional Cases

When validating parameters, consider all possible combinations:

- Valid parameters: Process normally (including boundary cases like empty lists)
- Invalid parameters: Throw an exception
    - Use `IllegalArgumentException` for invalid arguments
    - Example: `throw new IllegalArgumentException("error message");`

Testing with Exceptions

JUnit 5 provides ways to test exception handling:

```
@Test
void testExpectedException() {
    Assertions.assertThrows(IllegalArgumentException.class,
        () -> { new RationalNumber2(2,0);} );
}
```

try-catch testing:

When testing exception handling with try-catch blocks, we can verify that exceptions are thrown under the expected conditions and handle them appropriately. This approach provides a more detailed view of the exception handling process compared to JUnit assertions, as we can include custom messages and multiple catch blocks if needed.

```
try {
    test1 = new RationalNumber2(2, 0);
    System.out.println("Creation test failed");
} catch (IllegalArgumentException ex) {
    System.out.println("Creation test worked");
}
```

Class and Object Identification

Key principles for identifying classes in object-oriented design:

- Classes can come from:
    - Existing class libraries
    - Previous projects (reuse)
    - New implementations

Objects are typically represented by nouns, while their services (methods) are represented by verbs.

7.2 Identifying Classes and Objects

When designing object-oriented software, we need to determine which classes will make up our program. This is a crucial first step in representing the solution's elements.

Finding Potential Classes

- Examine the problem carefully
- Review functional specifications if available
- Look for **nouns** in program requirements

Of course, not every noun becomes a class. Remember that classes represent groups of objects with similar behaviour. You must decide whether to represent each element as an object or a primitive representation

For example: should employee salary be a float in the Employee class or a separate Salary class? Either could be correct, depending on the program you're developing.

Support Classes

While identifying classes representing the 'nouns' of our program is important, we also need to consider classes that provide services or support functionality. These supporting classes are crucial for organizing code, maintaining separation of concerns, and ensuring the application runs smoothly even though they might not directly represent objects from the problem domain.

Don't Reinvent the Wheel

During real system development, some needed classes may already exist. These could be from the Java standard library, previous solutions, or third-party libraries - all forms of software reuse. Even if an existing class isn't an exact match, it may serve as a foundation for a new class.

Assigning Responsibilities

When identifying classes, always consider their responsibilities. Classes represent objects with behaviours defined by their methods. Class behaviours define program functionality, so we use **verbs** to name them. Early design stages don't require identifying all methods - focus on primary responsibilities and how they

translate to methods. If no existing class is appropriate for a required behaviour, this can be a sign that a new class is needed.

7.3 Static Class Members

A class member marked with the `static` keyword belongs to the class itself. Static members are shared among all instances of the class and can be accessed without creating a class instance.

Static Variables

Static variables (aka class variables) are initialized when the class is first loaded, and shared across all instances of a class. Unlike instance variables where each object has its own copy, static variables have only one copy that belongs to the class itself.

```
public class Counter {

    private static int count = 0;  // Shared by all Counter objects



    public Counter() {

        count++;  // Increments the shared counter

    }

}
```

Constants (`final` variables) are often declared `static`, since there may as well be only one copy of an immutable value.

Static Methods

Static methods (aka class methods) are called through the class name rather than an object instance. We've already seen examples of static methods in the Math class. Recall how we did not need an instance of Math to use methods such as `Math.sqrt`:

```
Math.sqrt(27);  // Called directly through class name
```

Static methods cannot reference instance methods/variables within their class- only other static members. For example:

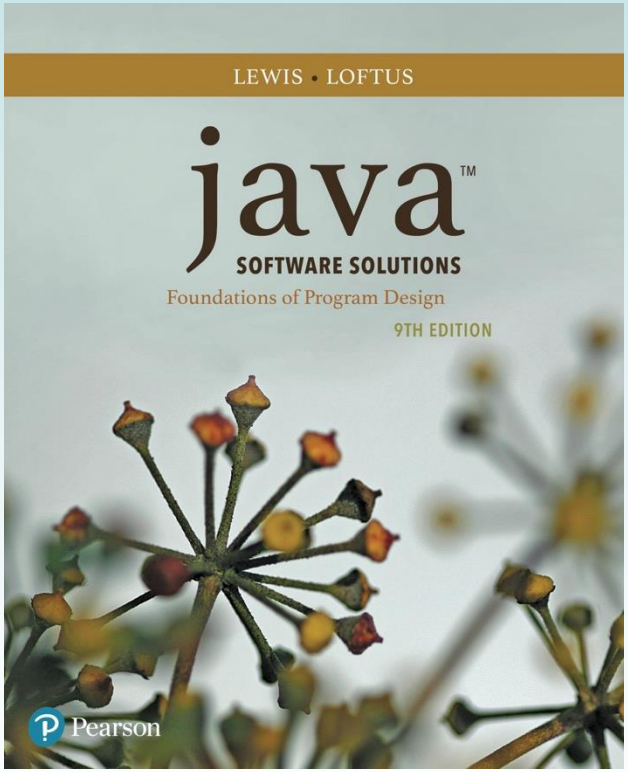
```
public class Example {

    private int instanceVar = 0;
```

```
    private static int staticVar = 0;


    public static void staticMethod() {
        staticVar = 1;     // OK - can access static variables
        instanceVar = 1;   // ERROR - cannot access instance variables
    }
}
```

Chapter 7
Object-Oriented Design



Java Software Solutions

Foundations of Program Design

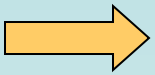
9th Edition

John Lewis
William Loftus

Object-Oriented Design

- Now we can extend our discussion of the design of classes and objects
- Chapter 7 focuses on:
  - software development activities
  - the relationships that can exist among classes
  - the static modifier
  - writing interfaces
  - the design of enumerated type classes
  - method design and method overloading
  - GUI design
  - mouse and keyboard events

Outline



Software Development Activities

Static Variables and Methods

Class Relationships

Interfaces

Enumerated Types Revisited

Method Design and Overloading

GUI Design

Mouse and Keyboard Events

Program Development

- The creation of software involves four basic activities:
  - establishing the requirements
  - creating a design
  - implementing the code
  - testing the implementation
- These activities are not strictly linear – they overlap and interact

Requirements

- *Software requirements* specify the tasks that a program must accomplish
  - what to do, not how to do it
- Often an initial set of requirements is provided, but they should be critiqued and expanded
- It is difficult to establish detailed, unambiguous, and complete requirements
- Careful attention to the requirements can save significant time and expense in the overall project

Design

- A *software design* specifies how a program will accomplish its requirements
- A software design specifies how the solution can be broken down into manageable pieces and what each piece will do
- An object-oriented design determines which classes and objects are needed, and specifies how they will interact
- Low level design details include how individual methods will accomplish their tasks

Implementation

- *Implementation* is the process of translating a design into source code
- Novice programmers often think that writing code is the heart of software development, but actually it should be the least creative step
- Almost all important decisions are made during requirements and design stages
- Implementation should focus on coding details, including style guidelines and documentation

Testing

- *Testing* attempts to ensure that the program will solve the intended problem under all the constraints specified in the requirements
- A program should be thoroughly tested with the goal of finding errors
- *Debugging* is the process of determining the cause of a problem and fixing it
- Debugging and testing were covered in the last chapter

Validating Parameters

- Often methods have restrictions on parameter values
  - Integer.parseInt(String arg) requires the argument to have the form of an integer
  - Math.sqrt(double x) returns the positive square root of a double x which should be positive
- If the parameter is not valid, handle it properly
  - Integer.parseInt throws NumberFormatException if the argument is not in the form of an integer
  - Math.sqrt returns NaN if the argument < zero

Handling Exceptional Cases

- Consider what to do for all possible combinations of the parameters
- Possibilities:
  - If the parameters represent a valid combination, process them normally
    - includes boundary cases such as processing a list of zero items
  - If the parameters are invalid, throw an exception
    - `IllegalArgumentException` for now
    - indicates that a method has been passed an illegal or inappropriate argument

```
throw new IllegalArgumentException("error message");
```

Testing with Exceptions

- Exceptions are covered in chapter 11
- There is a new statement that allows a program to continue (not crash) when it gets an exception

```
try {
     test1 = new RationalNumber2(2, 0);
     System.out.println("Creation test failed");
} catch (IllegalArgumentException ex) {
     System.out.println("Creation test worked");
}
```

- To test with JUnit 5 assertThrows use Lambda expression:

```
@Test
void testExpectedException() {
  Assertions.assertThrows(IllegalArgumentException.class,
          () -> { new RationalNumber2(2,0);} );
}
```

Identifying Classes and Objects

- The core activity of object-oriented design is determining the classes and objects that will make up the solution
- The classes may be part of a class library, reused from a previous project, or newly written
- One way to identify potential classes is to identify the objects discussed in the requirements
- Objects are generally nouns, and the services that an object provides are generally verbs

Identifying Classes and Objects

- A partial requirements document:

> The user must be allowed to specify each product by its primary characteristics, including its name and product number. If the bar code does not match the product, then an error should be generated to the message window and entered into the error log. The summary report of all transactions must be structured as specified in section 7.A.

- Of course, not all nouns will correspond to a class or object in the final solution

Identifying Classes and Objects

- Remember that a class represents a group (classification) of objects with the same behaviors
- Generally, classes that represent objects should be given names that are singular nouns
- Examples: `Coin`, `Student`, `Message`
- A class represents the concept of one such object
- We are free to instantiate as many of each object as needed

Identifying Classes and Objects

- Sometimes it is challenging to decide whether something should be represented as a class
- For example, should an employee's address be represented as a set of instance variables or as an `Address` object
- The more you examine the problem and its details the more clear these issues become
- When a class becomes too complex, it often should be decomposed into multiple smaller classes to distribute the responsibilities

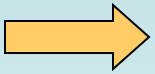
Identifying Classes and Objects

- We want to define classes with the proper amount of detail
- For example, it may be unnecessary to create separate classes for each type of appliance in a house
- It may be sufficient to define a more general `Appliance` class with appropriate instance data
- It all depends on the details of the problem being solved

Identifying Classes and Objects

- Part of identifying the classes we need is the process of *assigning responsibilities* to each class
- Every activity that a program must accomplish must be represented by one or more methods in one or more classes
- We generally use verbs for the names of methods
- In early stages it is not necessary to determine every method of every class – begin with primary responsibilities and evolve the design

Outline



Software Development Activities

Static Variables and Methods

Class Relationships

Interfaces

Enumerated Types Revisited

Method Design and Overloading

GUI Design

Mouse and Keyboard Events

Static Class Members

- Recall that a static method is one that can be invoked through its class name
- For example, the methods of the `Math` class are static:

```
result = Math.sqrt(25)
```

- Variables can be static as well
- Determining if a method or variable should be static is an important design decision

The static Modifier

- We declare static methods and variables using the `static` modifier
- It associates the method or variable with the class rather than with an object of that class
- Static methods are sometimes called *class methods* and static variables are sometimes called *class variables*
- Let's carefully consider the implications of each

Static Variables

- Normally, each object has its own data space, but if a variable is declared as static, only one copy of the variable exists

```
private static float price;
```

- Memory space for a static variable is created when the class is first referenced
- All objects instantiated from the class share its static variables
- Changing the value of a static variable in one object changes it for all others

Static Methods

```
public class Helper
{
    public static int cube(int num)
    {
        return num * num * num;
    }
}
```

- Because it is declared as static, the `cube` method can be invoked through the class name:

```
value = Helper.cube(4);
```

Static Class Members

- The order of the modifiers can be interchanged, but by convention visibility modifiers come first
- Recall that the `main` method is static – it is invoked by the Java interpreter without creating an object
- Static methods cannot reference instance variables because instance variables don't exist until an object exists
- However, a static method can reference static variables or local variables

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`
- See `Slogan.java`

Quick Check

Why can't a static method refer to an instance variable?

Quick Check

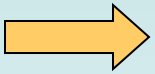
Why can't a static method refer to an instance variable?

Because instance data is created only when an object is created.

You don't need an object to execute a static method.

And even if you had an object, which object's instance data would be referenced? (remember, the method is invoked through the class name)

Outline



Software Development Activities

Static Variables and Methods

Class Relationships

Interfaces

Enumerated Types Revisited

Method Design and Overloading

GUI Design

Mouse and Keyboard Events

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 9

Dependency

- A *dependency* exists when one class relies on another in some way, usually by invoking the methods of the other
- We've seen dependencies in many previous examples
- We don't want numerous or complex dependencies among classes
- Nor do we want complex classes that don't depend on others
- A good design strikes the right balance

Dependency

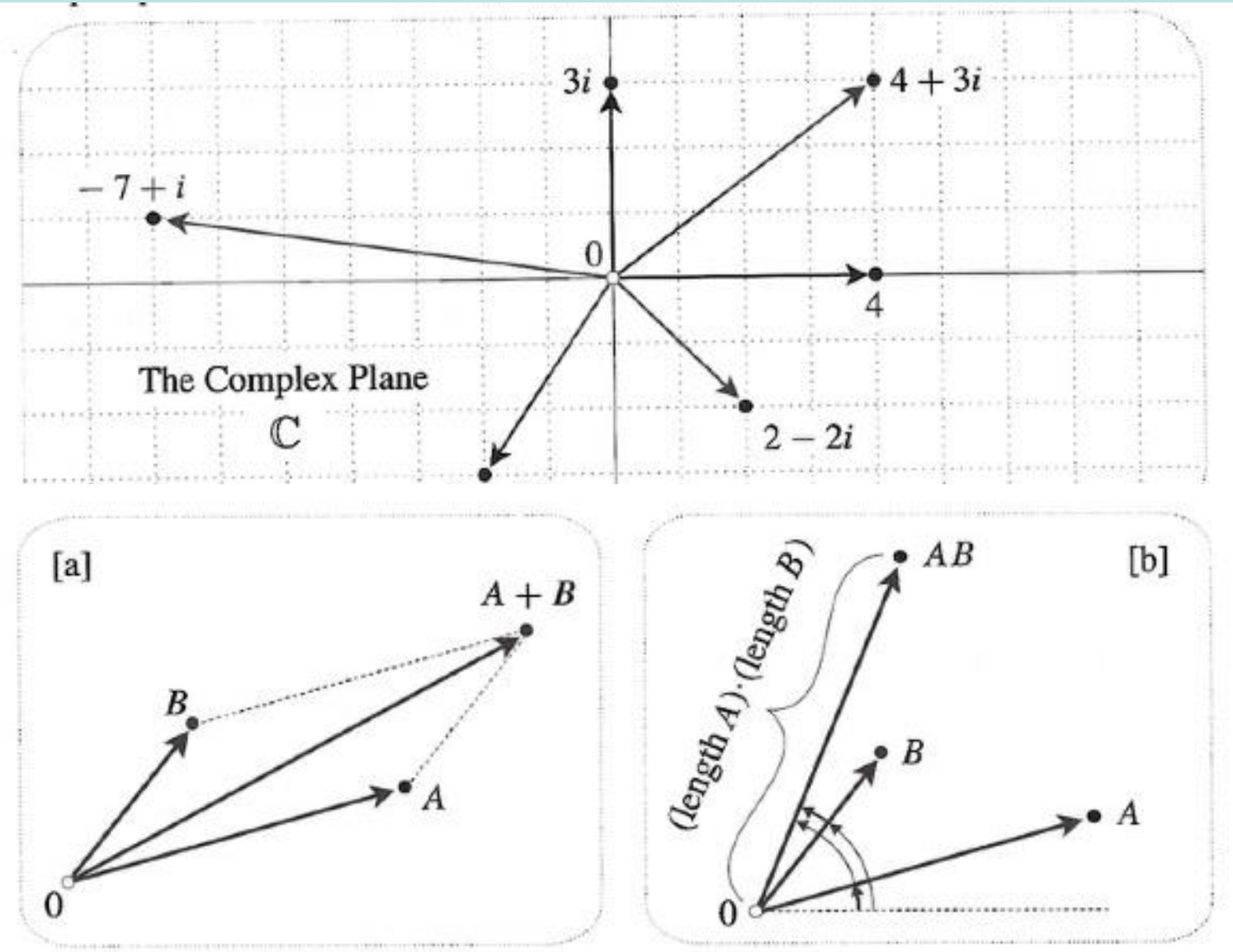
- Some dependencies occur between objects of the same class
- A method of the class may accept an object of the same class as a parameter
- For example, the `concat` method of the `String` class takes as a parameter another `String` object

```
str3 = str1.concat(str2);
```

Dependency

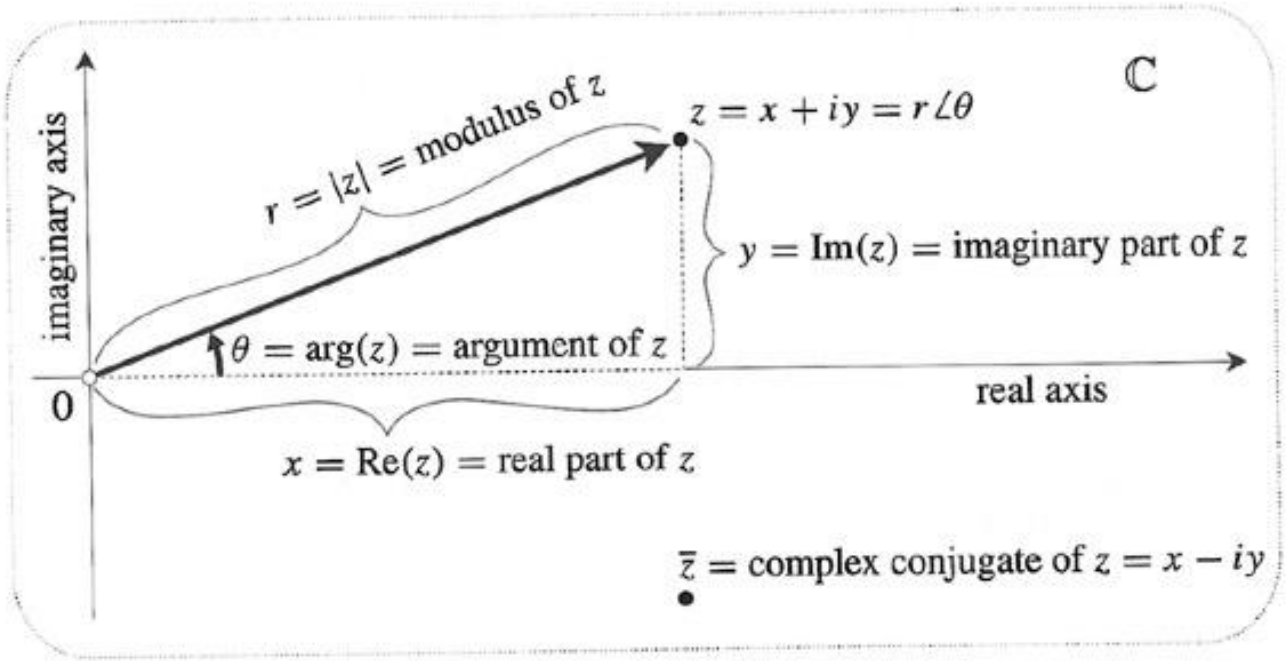
- The following example defines a class called `Complex` that represents a complex number
- A complex number is a value that can be represented as two doubles
  - Called the real and imaginary part
- Several methods of the `Complex` class accept another `Complex` object as a parameter
- See `Complex.java`
- See `ComplexTester.java`

Complex Numbers



Complex Numbers, Definitions

Name	Meaning	Notation
modulus of z	length r of z	$\lvert z\rvert$
argument of z	angle θ of z	$\arg(z)$
real part of z	x coordinate of z	$\mathrm{Re}(z)$
imaginary part of z	y coordinate of z	$\mathrm{Im}(z)$
imaginary number	real multiple of i	
real axis	set of real numbers	
imaginary axis	set of imaginary numbers	
complex conjugate of z	reflection of z in the real axis	$\bar{z}$



equals() and hashCode()

- Example properly implements the equals() and hashCode() methods
- equals indicates whether some other object is logically the same as this one
- equals should be
  - reflexive: x.equals(x) should return true,
  - symmetric: x.equals(y) == y.equals(x) and
  - transitive: if x.equals(y) returns true and y.equals(z) returns true then x.equals(z) should return true
- x.equals(null) should return false
- Default x.equals(y) returns true iff x, y are aliases

hashCode()

- hashCode() returns an int value that is useful in implementing hash tables
- hashCode must return the same value if the information used in equals() has not changed
- if two objects are equal according to equals() then calling hashCode on each object must return the same result
- As much as possible, if x.equals(y) returns false, then hashCode should return different values for x and y

Example of equals and hashCode

```
public boolean equals(Object op2) {

   return (op2 instanceof Complex c
      && re == c.re
      && im == c.im);
}



public int hashCode() {

   return Double.hashCode(re)
        ^  Double.hashCode(im);
}
```

Note: if op2 is null, instanceof is false.
If op2 cannot be cast to Complex, instanceof is false.
If op2 can be cast to Complex, then it is and its reference is stored in c.

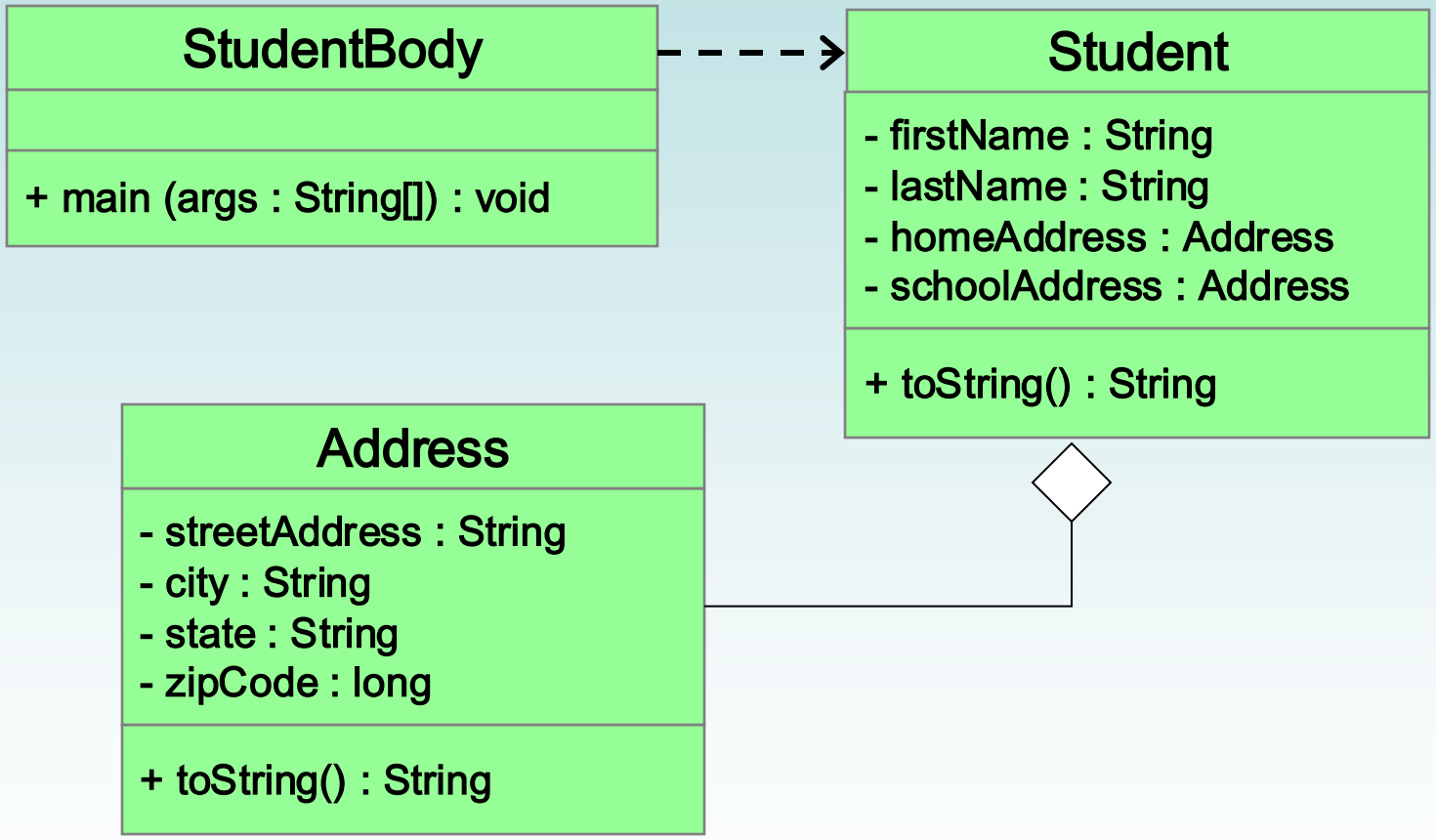
Aggregation

- An *aggregate* is an object that is made up of other objects
- Therefore aggregation is a *has-a* relationship
  - A car *has a* chassis
- An aggregate object contains references to other objects as instance data
- This is a special kind of dependency; the aggregate relies on the objects that compose it

Aggregation

- In the following example, a `Student` object is composed, in part, of `Address` objects
- A student has an address (in fact each student has two addresses)
- See `StudentBody.java`
- See `Student.java`
- See `Address.java`

Aggregation in UML



The this Reference

- The `this` reference allows an object to refer to itself
- That is, the `this` reference, used inside a method, refers to the object through which the method is being executed
- Suppose the `this` reference is used inside a method called `tryMe`, which is invoked as follows:

```
obj1.tryMe();
obj2.tryMe();
```

- In the first invocation, the `this` reference refers to `obj1`; in the second it refers to `obj2`

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:

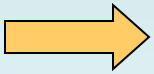
```
public Account(String name, long acctNumber,
               double balance)
{
   this.name = name;
   this.acctNumber = acctNumber;
   this.balance = balance;
}
```

Outline

Software Development Activities

Static Variables and Methods

Class Relationships



Interfaces

Enumerated Types Revisited

Method Design and Overloading

GUI Design

Mouse and Keyboard Events

Interfaces

- A Java *interface* is a collection of abstract methods, static methods, default 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 usually it is left off
- An interface is used to establish a set of methods that a class will implement

Default Methods

- Default methods allow adding a method to an API without breaking old code
- Allows evolving old interfaces without breaking old code
- Allows adding convenience methods
- Uses reserved word `default`
- Example:

```
public interface Collection {
   int size();
   default boolean isEmpty() {
      return size() == 0;
   }
   …
}
```

Interfaces

`interface` is a reserved word

None of the methods in an interface are given a definition (body)

A semicolon immediately follows each method header

```
public interface Doable {
    int MIN_VALUE = 23;
    void doThis();
    int doThat();
    void doThis2(double value, char ch);
    boolean doTheOther(int num);
    default void remove() {
        throw new UnsupportedOperationException();
    }

}
```

Interfaces

- An interface cannot be instantiated
- Methods in an interface have public visibility by default
- A class formally implements an interface by:
  - stating so in the class header
  - providing implementations for every abstract method in the interface
- If a class declares that it implements an interface, it must define *all* abstract methods in the interface

Interfaces

`implements` is a reserved word

```
public class CanDo implements Doable
{
    public void doThis()
    {
        // whatever
    }

    public void doThat()
    {
        // whatever
    }

    // etc.
}
```

Each method listed in `Doable` is given a definition

Interfaces

- 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
- A class that implements an interface can implement other methods as well
- See `Complexity.java`
- See `Question.java`
- See `MiniQuiz.java`

Multiple Interfaces

- A class can implement multiple interfaces
- The interfaces are listed in the `implements` clause
- The class must implement all abstract methods in all interfaces listed in the header
- A class can get the same default method from multiple interfaces
  - But it must then provide its own implementation of it

```
class ManyThings implements interface1, interface2
{
   // all methods of both interfaces
}
```

Java Standard Interfaces

- The Java API contains many helpful interfaces
- The `Comparable` interface contains one abstract method called `compareTo`, which is used to compare two objects
- We discussed the `compareTo` method of the `String` class in Chapter 5
- The `String` class implements `Comparable`, giving us the ability to put strings in lexicographic order

The Comparable Interface

- Any class can implement `Comparable` to provide a mechanism for comparing objects of that type

```
if (obj1.compareTo(obj2) < 0)
   System.out.println ("obj1 is less than obj2");
```

- The value returned from `compareTo` should be
  - negative if `obj1` is less that `obj2`,
  - 0 if they are equal, and
  - positive if `obj1` is greater than `obj2`
- It's up to the programmer to determine what makes one object less than another

The Iterator Interface

- As discussed in Chapter 5, an iterator is an object that provides a means of processing a collection of objects one at a time
- An iterator is created formally by implementing the `Iterator` interface, which contains two abstract methods
  - The `hasNext` method returns a boolean result – true if there are items left to process
  - The `next` method returns the next object in the iteration
- There are also two default methods
  - `remove` and `forEachRemaining`

The Iterable Interface

- Another interface, `Iterable`, establishes that an object provides an iterator
- The `Iterable` interface has one method, `iterator`, that returns an `Iterator` object
  - There are also two default methods added for functional programming (beyond the scope of this course)
- Any `Iterable` object can be processed using the for-each version of the `for` loop
- Note the difference: an `Iterator` has methods that perform an iteration; an `Iterable` object provides an iterator on request

Interfaces - Conclusion

- You could write a class that implements certain methods (such as `compareTo`) without formally implementing the interface (`Comparable`)
- However, formally establishing the relationship between a class and an interface allows Java to deal with an object in certain ways
- Interfaces are a key aspect of object-oriented design in Java
- This idea is discussed further in Chapter 10
  - See next Java course

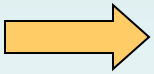
Outline

Software Development Activities

Static Variables and Methods

Class Relationships

Interfaces



Enumerated Types Revisited

Method Design and Overloading

GUI Design

Mouse and Keyboard Events

Enumerated Types

- In Chapter 3 we introduced enumerated types, which define a new data type and list all possible values of that type:

```
enum Season {winter, spring, summer, fall}
```

- Once established, the new type can be used to declare variables

Season time;

- The only values this variable can be assigned are the ones established in the `enum` definition

Enumerated Types

- An enumerated type definition is a special kind of class
- The values of the enumerated type are objects of that type
- For example, `fall` is an object of type `Season`
- That's why the following assignment is valid:

```
time = Season.fall;
```

Enumerated Types

- An enumerated type definition can be more interesting than a simple list of values
- Because they are like classes, we can add additional instance data and methods
- We can define an `enum` constructor as well
- Each value listed for the enumerated type calls the constructor
- See `Season.java`
- See `SeasonTester.java`

Enumerated Types

- Every enumerated type contains a static method called `values` that returns a list of all possible values for that type
- The list returned from `values` can be processed using a for-each loop
- An enumerated type cannot be instantiated outside of its own definition
- A carefully designed enumerated type provides a versatile and type-safe mechanism for managing data

Outline

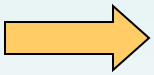
Software Development Activities

Static Variables and Methods

Class Relationships

Interfaces

Enumerated Types Revisited



Method Design and Overloading

GUI Design

Mouse and Keyboard Events

Method Design

- As we've discussed, high-level design issues include:
  - identifying primary classes and objects
  - assigning primary responsibilities
- After establishing high-level design issues, its important to address low-level issues such as the design of key methods
- For some methods, careful planning is needed to make sure they contribute to an efficient and elegant system design

Method Decomposition

- A method should be relatively small, so that it can be understood as a single entity
- A potentially large method should be decomposed into several smaller methods as needed for clarity
- A public service method of an object may call one or more private support methods to help it accomplish its goal
- Support methods might call other support methods if appropriate

Method Decomposition

- Let's look at an example that requires method decomposition – translating English into Pig Latin
- Pig Latin is a language in which each word is modified by moving the initial sound of the word to the end and adding "ay"
- Words that begin with vowels have the "yay" sound added on the end

book → ookbay

table → abletay

item → itemyay

chair → airchay

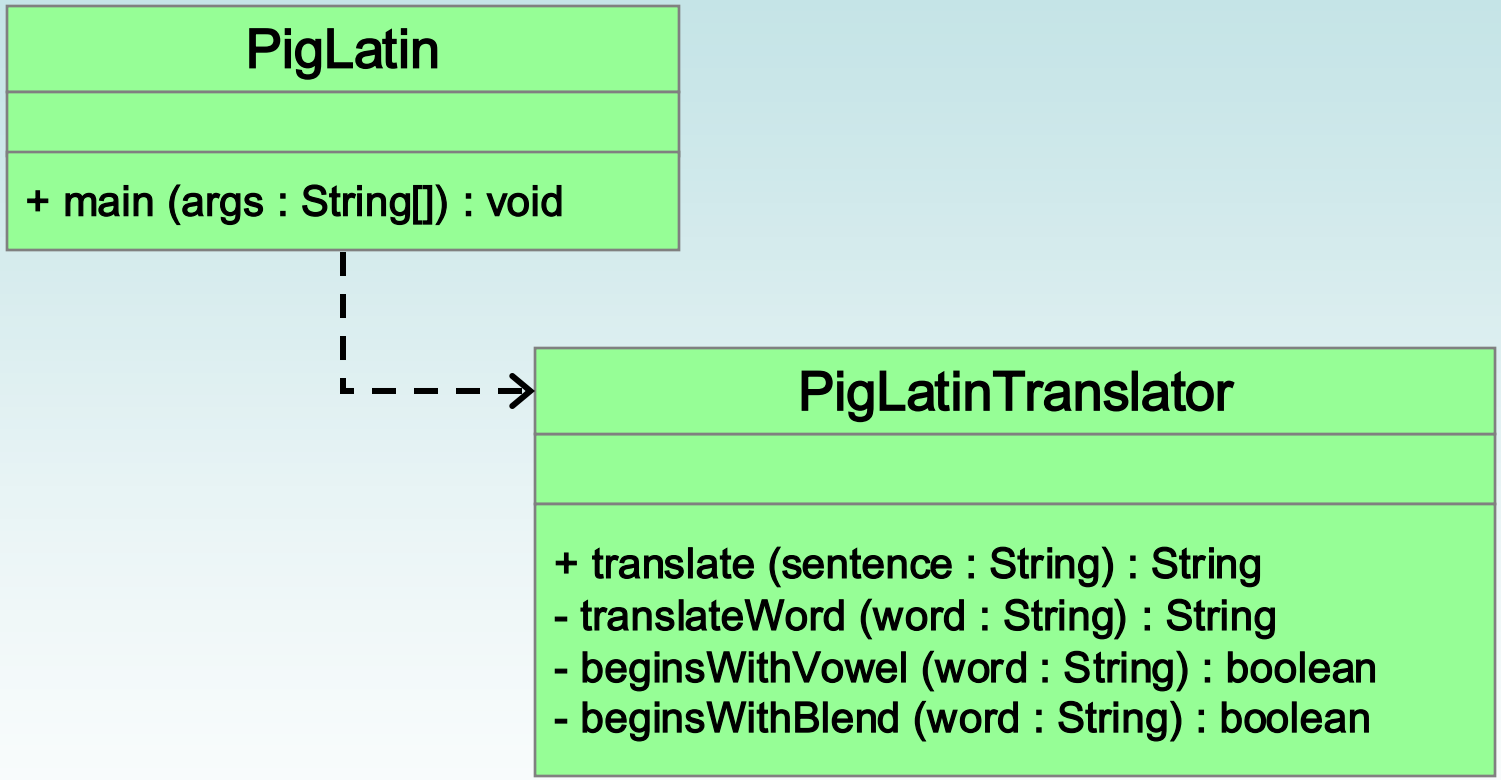
Method Decomposition

- The primary objective (translating a sentence) is too complicated for one method to accomplish
- Therefore we look for natural ways to decompose the solution into pieces
- Translating a sentence can be decomposed into the process of translating each word
- The process of translating a word can be separated into translating words that:
  - begin with vowels
  - begin with consonant blends (sh, cr, th, etc.)
  - begin with single consonants

Method Decomposition

- In a UML class diagram, the visibility of a variable or method can be shown using special characters
- Public members are preceded by a plus sign
- Private members are preceded by a minus sign
- See `PigLatin.java`
- See `PigLatinTranslator.java`

Class Diagram for Pig Latin



Objects as Parameters

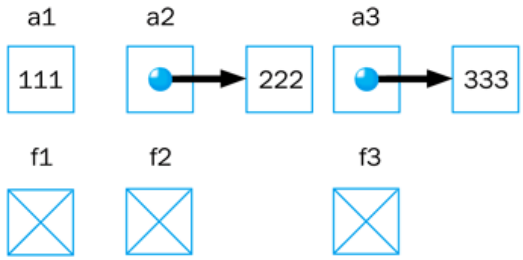
- Another important issue related to method design involves parameter passing
- Parameters in a Java method are *passed by value*
- A copy of the *actual parameter* (the value passed in) is stored into the *formal parameter* (in the method header)
- When an object is passed to a method, the actual parameter 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)
- Note the difference between changing the internal state of an object versus changing which object a reference points to
- See `ParameterTester.java`
- See `ParameterModifier.java`
- See `Num.java`

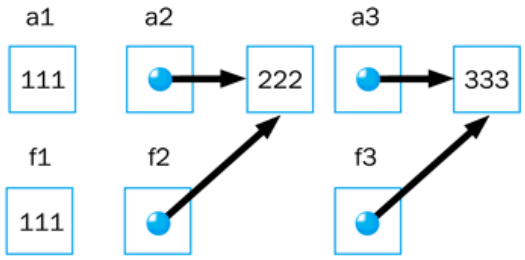
STEP 1

Before invoking `changeValues`



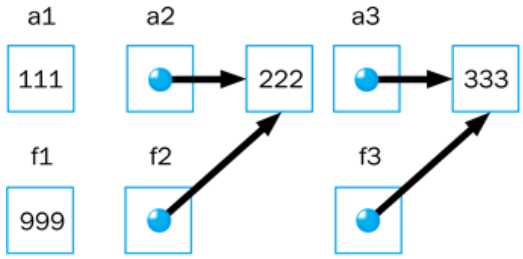
STEP 2

`tester.changeValues (a1, a2, a3);`



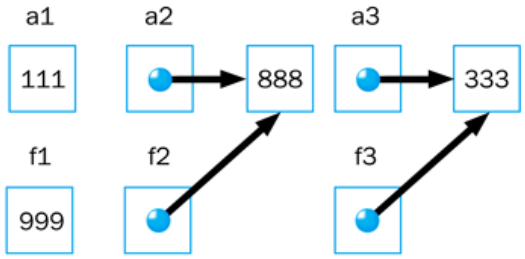
STEP 3

`f1 = 999;`



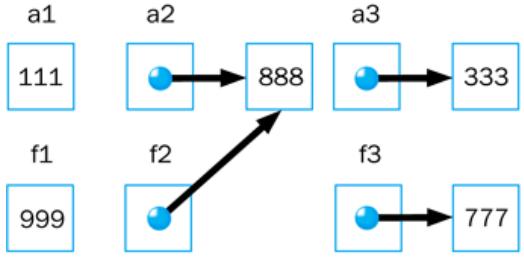
STEP 4

`f2.setValue (888);`



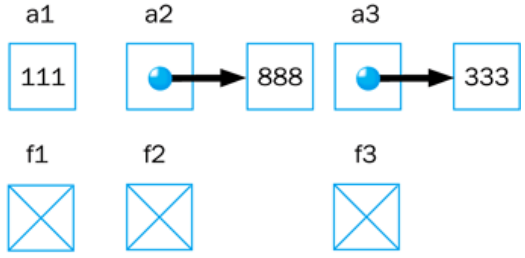
STEP 5

`f3 = new Num (777);`



STEP 6

After returning from `changeValues`



Method Overloading

- Let's look at one more important method design issue: method overloading
- *Method overloading* is the process of giving a single method name multiple definitions in a class
- 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

Method Overloading

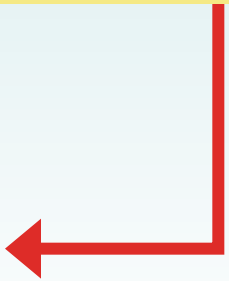
- The compiler determines which method is being invoked by analyzing the parameters

```
float tryMe(int x)
{
   return x + .375;
}

float tryMe(int x, float y)
{
   return x*y;
}
```

Invocation

```
result = tryMe(25, 4.32)
```



Method Overloading

- The `println` method is overloaded:

```
println(String s)
println(int i)
println(double d)
```

and so on...

- The following lines invoke different versions of the `println` method:

```
System.out.println("The total is:");
System.out.println(total);
```

Overloading Methods

- The return type of the method is not part of the signature
- That is, overloaded methods cannot differ only by their return type
- Constructors can be overloaded
- Overloaded constructors provide multiple ways to initialize a new object

Outline

Software Development Activities

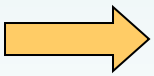
Static Variables and Methods

Class Relationships

Interfaces

Enumerated Types Revisited

Method Design and Overloading



GUI Design

Mouse and Keyboard Events

GUI Design

- We must remember that the goal of software is to help the user solve the problem
- To that end, the GUI designer should:
  - Know the user
  - Prevent user errors
  - Optimize user abilities
  - Be consistent
- Let's discuss each of these in more detail

Know the User

- Knowing the user implies an understanding of:
  - the user's true needs
  - the user's common activities
  - the user's level of expertise in the problem domain and in computer processing
- We should also realize these issues may differ for different users
- Remember, to the user, the interface is the program

Prevent User Errors

- Whenever possible, we should design user interfaces that minimize possible user mistakes
- We should choose the best GUI components for each task
- For example, in a situation where there are only a few valid options, using a menu or radio buttons would be better than an open text field
- Error messages should guide the user appropriately

Optimize User Abilities

- Not all users are alike – some may be more familiar with the system than others
- Knowledgeable users are sometimes called *power users*
- We should provide multiple ways to accomplish a task whenever reasonable
  - "wizards" to walk a user through a process
  - short cuts for power users
- Help facilities should be available but not intrusive

Be Consistent

- Consistency is important – users get used to things appearing and working in certain ways
- Colors should be used consistently to indicate similar types of information or processing
- Screen layout should be consistent from one part of a system to another
- For example, error messages should appear in consistent locations

Outline

Software Development Activities

Static Variables and Methods

Class Relationships

Interfaces

Enumerated Types Revisited

Method Design and Overloading

GUI Design

Mouse and Keyboard Events

Mouse Events

- JavaFX nodes can generate several types of mouse-based events:

Event	Description
mouse pressed	mouse button is pressed
mouse released	mouse button is released
mouse clicked	mouse button is pressed and released
mouse entered	mouse pointer is moved onto a node
mouse exited	mouse is moved off of a node
mouse moved	mouse is moved
mouse dragged	mouse is moved while holding the mouse button down

Mouse Events

- The `MouseEvent` object representing the event can be used to obtain the mouse position
- There are convenience methods for setting the handler for each type of mouse event (such as `setOnMousePressed`)
- See `ClickDistance.java`

Mouse Events

- A stream of mouse moved or mouse dragged events occur while the mouse is in motion
- This essentially allows the program to track the movement in real time
- Using the mouse to "draw" a shape into place is called *rubberbanding*
- See `RubberLines.java`

Key Events

- There are three JavaFX events related to the user typing at the keyboard:

Event	Description
key pressed	a keyboard key is pressed down
key released	a keyboard key is released
key typed	a keyboard key that generates a character is typed (pressed and released)

- The `getCode` method of the event object returns a code that represents the key that was pressed
- See `AlienDirection.java`

Summary

- Chapter 7 has focused on:
  - software development activities
  - the relationships that can exist among classes
  - the static modifier
  - writing interfaces
  - the design of enumerated type classes
  - method design and method overloading
  - GUI design
  - mouse and keyboard events