COMP 401

Prasun Dewan[1]

21. More Inheritance

In this chapter, we will go deeper into inheritance. We will distinguish between bottom-up design of inheritance, which we did in the examples of chapter 17, and top-down design of type hierarchy, which we will illustrate in this chapter. We will also try and understand the order in which variable initialization and constructor invocation occurs in a class hierarchy. We will see that in addition to constructors, it is useful to have init methods for initializing variables. We will understand the new concepts of abstract classes and methods and how they differ from the related concept of interfaces. Furthermore, we will explicitly look at the notion of dynamic dispatch, which unlike overloading, resolves method calls at execution time. Finally, the example we use here will allow us to detour into a quick overview of grammars, parsing, and recursive-descent parsers.

Factoring classes

To help illustrate these additional concepts, let us consider the example in Figure 24. The program allows a user to enter a list of course titles terminated by a string consisting of the period character. As each course title is entered, it determines if the course is offered. If the course is not offered, it prints out a message indicating this. Otherwise, it prints the name of the department offering the course, its number, and the title.

The program must search some collection of courses to answer the user queries. We will assume this list is hardwired into the program (rather than being input by somebody or read from a file) and is stored in some special collection class we create. The list would provide a method to add a course objects and another that determines if a course corresponding to a specified title exists. The main program would crate the collection and call its methods. Let us now look deeply at the nature of these the course objects added to the collection.

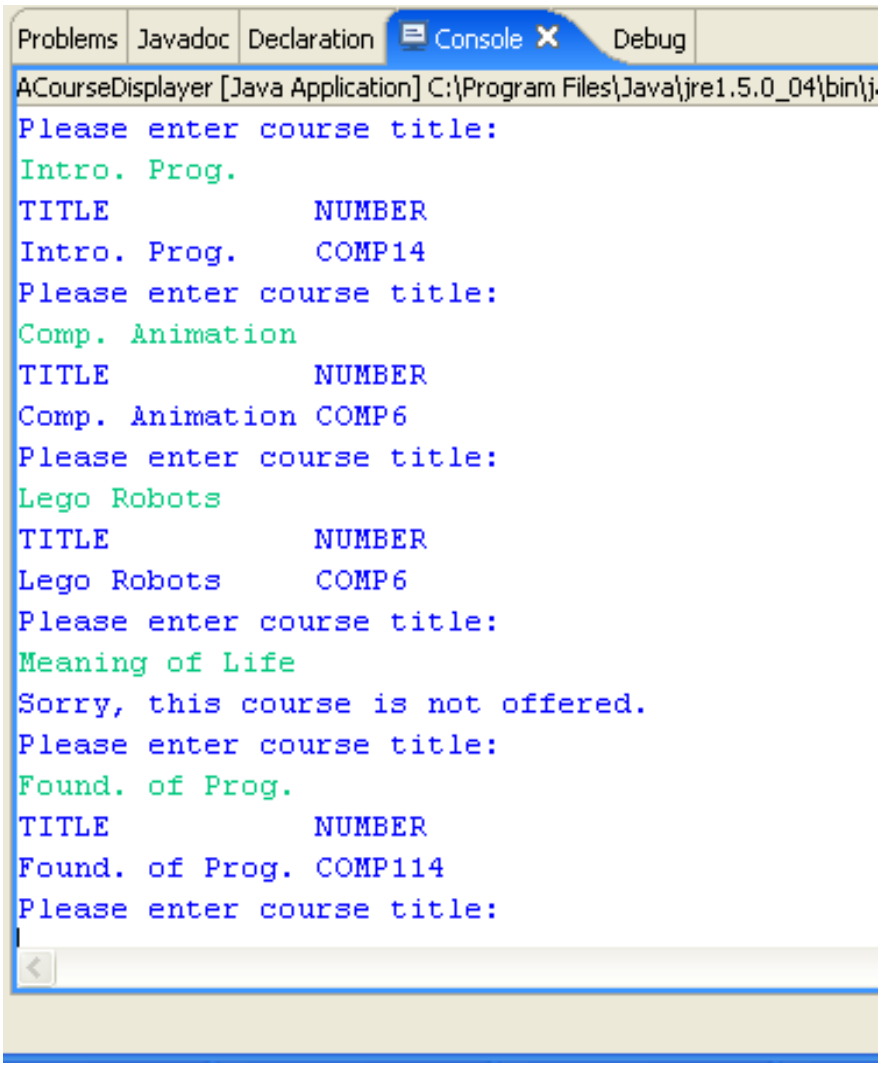


Figure 1 Course Displayer

One alternative would be to create a single class, ACourse, for all course objects with properties for the course title, number, and department. Each of these properties would be stored in a separate variable. While suitable for a regular course, such a class is not suitable for a freshman seminar. The reason is, as indicated by the user interface above, all freshman seminars have the same number. Thus, it can be constant rather than a variable in a freshman seminar object.

Another alternative is to create two different classes, ARegularCourse and AFreshmanSeminar, for the two kinds of courses. Both classes would have properties for the course title, number, and department. Moreover, both would have instance variables for the title and department. The difference would be that one class would create a variable for the number while the other would create a named constant.

A problem with this approach is that code associated with implementing the title and department properties is not shared by the two classes. The answer, then, is to create a third class, A Course, for implementing these two properties, and modify the original two classes so that they inherit this code.

Thus, we see here another use of inheritance. However, the process used to create the class hierarchy here is different from the one we used before. Earlier, we used an opportunistic, top-down strategy – we had the class AStringHistory, and we needed to create AStringDatabase or AStringSet, we simply inherited from an *existing* class. Here, however, we created two independent classes first, and then discovered in a bottom-up approach that their shared code can be factored into a common superclass. Thus, we did not simply reuse an existing class created for some other problem. We created the superclass especially for this problem by factoring common code in the subclases.

Abstract class

In a bottom-up approach, the superclasses we create may not be first-class classes in that they may not be instantiatable. This is the case in this example. It does not make sense to instantiate the superclass, ACourse, since it does not have a property for a course number. All courses in our world are required to have course numbers. Thus, ACourse defines part of the behavior of an object (of some subclass), it does not by itself define a complete object. Such a class is called an abstract class.

The difference between an abstract and regular class is similar to the difference between an animal species such as homo sapiens and a species group such as mammal. A species describes the traits of actual animals while a species group describes traits shared by the member species but does not describe a complete animal. Similarly, we cannot simply buy a car or grow a plant, we must buy a specific car or grow a specific plant, even though the terms *car* and *plant* help us classify objects of more specific classes. Classes that are not bstract are called *concrete classes.*

The following code shows how an abstract class is declared:

```
package courses;
public abstract class ACourse  {
        String title, dept;
        public ACourse (String theTitle, String theDept) {
                title = theTitle;
                dept = theDept;
        }
        public String getTitle() {
                return title;
        }
        public String getDepartment() {
                return dept;
        }
}
```

The keyword, **abstract**, appears in the header of the class declaration. Java will not allow the program to instantiate a class declared as abstract. Even though this class will not instantiated, it defines a constructor, as it declares instance variables, which are initialized by the constructor.

An abstract class can be subclassed like a regular class, as shown in the two class declarations below:

```
package courses;
public class ARegularCourse extends ACourse implements Course {
        int courseNum;
        public ARegularCourse (String theTitle, String theDept, int theCourseNum) {
                super (theTitle, theDept);
                courseNum = theCourseNum;
        }
        public int getNumber() {
                return courseNum;
        }
}
```

```
package courses;
public class AFreshmanSeminar extends ACourse implements FreshmanSeminar {
        public AFreshmanSeminar (String theTitle, String theDept) {
                super (theTitle, theDept);
        }
        public int getNumber() {
                return SEMINAR_NUMBER;
        }
}
```

The class hierarchy we have created here is different from the one we created for the collection classes in several ways. One difference, of course, is that it uses abstract classes. In addition, the collection hierarchy was liner, with each class having exactly one direct subclass. A more subtle difference is that in this class hierarchy, variables are declared in both a class and its superclass. In the collection hierarchy, the two instance variables, contents and size, used by the three classes, AStringHistory, AStringDatabase, and AStringSet, were all declared in the topmost class, AStringHistory. In the example, ARegularCourse both declare its own variables and inherit them from ACourse. A final difference is that in the collection class hierarchy, no explicit constructors were defined to initialize the instance variables. In this hierarchy all classes define explicit constructors to initialize variables.

Calling superclass constructors

The constructor for AFreshmanSeminar does nothing but call the constructor of its superclass to initialize the inherited variables. The keyword **super** without a method name following it refers to the constructor of the superclass of the class in which the keyword appears. The constructor in ARegularCourse both initializes its variables and calls the constructor of the abstract class - it calls the constructor for the super class with the first two arguments, and then initializes its instance variable with the value of the third argument.. It is important to put this constructor – Java does not automatically call superclass.

What if the two statements in the constructor of ARegularCourse were switched?

```
public ARegularCourse (String theTitle, String theDept, int theCourseNum) {
                super (theTitle, theDept);
                courseNum = theCourseNum;
        }
```

Or what if the constructor of the superclass was not called:

```
public ARegularCourse (String theTitle, String theDept, int theCourseNum)
                courseNum = theCourseNum;
        }
```

The first alternative seems reasonable in our example but the second one is clearly wrong as the superclass variables are not initialized.

As it turns out, Java does not allow either of these alternatives. Java requires each constructor in class to call a constructor in its superclass in the first statement of the constructor. This ensures that variables in the superclass are initialized by its constructor before variables in the subclass are. As subclass variables can be assigned expressions involving superclass variables, this ensures that subclass variables are initialized to the expected values. Since superclass variables cannot refer to subclass variables, there is no danger in initializing the latter after the former. However, abstract methods, described below, do create such a danger.

How then is the constructor in our definition of ACourse valid?

```
        public ACourse (String theTitle, String theDept) {
                title = theTitle;
                dept = theDept;
        }
```

Recall that a class that does not explicitly extend any class extends Object, and every class has a programmer-defined or default constructor. Thus, the above rule implies that the constructor of ACourse should have called a constructor of Object in its first statement.

The reason the above definition is valid is that if a constructor does not explicitly call a superclass constructor, Java automatically inserts a call to the parameterless constructor:

```
public ACourse (String theTitle, String theDept) {
super(); // inserted by java in object code
        title = theTitle;
        dept = theDept;
}
```

Class Object has a parameterless constructor, so this definition is valid.

Then why was the second alternative definition of ARegularCourse constructor invalid?

```
public ARegularCourse (String theTitle, String theDept, int theCourseNum)
        courseNum = theCourseNum;
}
```

It is equivalent to:

```
public ARegularCourse (String theTitle, String theDept, int theCourseNum) {
        super();
        courseNum = theCourseNum;
}
```

Class ACourse does not have a parameterless constructor, so Java complains that it could not find such a constructor.

Completing the code

Let us continue with the example, giving remaining code required to complete it.

The interfaces of the two course classes are given below:

```
public interface Course {

    public String getTitle();

    public String getDepartment();

    public int getNumber();

}
```

```
package courses;

public interface FreshmanSeminar extends Course {
```

```
	public final int SEMINAR_NUMBER = 6;

}
```

The only purpose of creating the FreshmanSeminar interface is to declare the constant for SEMINAR_NUMBER. All other properties it inherits from Course – the interface of a regular course. Thus, both interfaces define the same set of methods.

The course interfaces and classes are used in the interfaces and classes defined for the course collection:

```
package collections;
import courses.Course;
public interface CourseList  {
	public void addElement(Course element);
	public Course matchTitle (String theTitle);
}

package collections;
import courses.Course;
public class ACourseList implements CourseList {
	final int MAX_SIZE = 50;
	Course[] contents = new Course[MAX_SIZE];
	int size = 0;
	int size() {
		return size;
	}
	boolean isFull() {
		return size == MAX_SIZE;
	}
	public void addElement(Course element) {
		if (isFull())
			System.out.println("Adding item to a full collection");
		else {
			contents[size] = element;
			size++;
		}
	}
	public Course matchTitle (String theTitle) {
		for (int courseIndex = 0; courseIndex < size; courseIndex++) {
			if (contents[courseIndex].getTitle().equals(theTitle))
				return contents[courseIndex];
		}
		return null;
	}
```

}

Recall that the String method, equals()checks if the target object contains the same sequence of characters as the parameter. You should always use this method for string comparison, since recall that == checks if two objects are the same (stored in the same memory location) while equals() check if their contents are the same.

As we saw before, the special value null indicates the absence of an object. This value is returned by matchTitle() if no course with the specified title can be found.

The main class uses both the collection class and the three course classes (and the associated interfaces:

```
package main;
import courses.Course;
import courses.ARegularCourse;
import courses.AFreshmanSeminar;
import collections.CourseList;
import collections.ACourseList;
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class ACourseDisplayer    {
        public static void main(String[] args)      {
                fillCourses();
                while (true) {
                        System.out.println("Please enter course title:");
                        String inputLine = readString();
                        if (inputLine.equals("."))
                                break;
                        Course matchedCourse = courses.matchTitle(inputLine);
                        if (matchedCourse == null)
                                System.out.println("Sorry, this course is not offered.");
                        else {
                                printHeader();
                                print (matchedCourse);
                        }
                }
        }
static CourseList courses = new ACourseList();
        static void fillCourses() {
                courses.addElement(new ARegularCourse ("Intro. Prog.",  "COMP", 14));
                courses.addElement(new ARegularCourse ("Found. of Prog.", "COMP", 114));
                        courses.addElement(new AFreshmanSeminar("Comp. Animation", "COMP"));
                                courses.addElement(new AFreshmanSeminar("Lego Robots",  "COMP"));
```

```
	}
static void printHeader() {
		System.out.println ("TITLE		"  + "NUMBER    ");
	}
	 static void print (Course course) {
		System.out.println(
				course.getTitle() + "	" +
				course.getDepartment() +
				course.getNumber()
				);

	}
	static BufferedReader dataIn = new BufferedReader (new InputStreamReader(System.in));
	static String readString() {
		try {
			return dataIn.readLine();
		} catch (Exception e) {
			return null;
		}
	}
}
```

As we see above, a program can use the null value in an == comparison.

Grammars and Recursive-Descent Parsing

Let us use this example to take a detour from inheritance to understand the important concept of parsing and grammars, which you need in your project. To do so, consider a variation of the course problem. Previously, the contents of the course list were hardwired in the main program. Now let us allow the user to input these, as shown in Figure 26. The console window now contains two parts separated by a line consisting of a period. The part is the pervious input, consisting of courses whose titles are to be matched. The first part is new, and contains the courses to be filled into the course database.

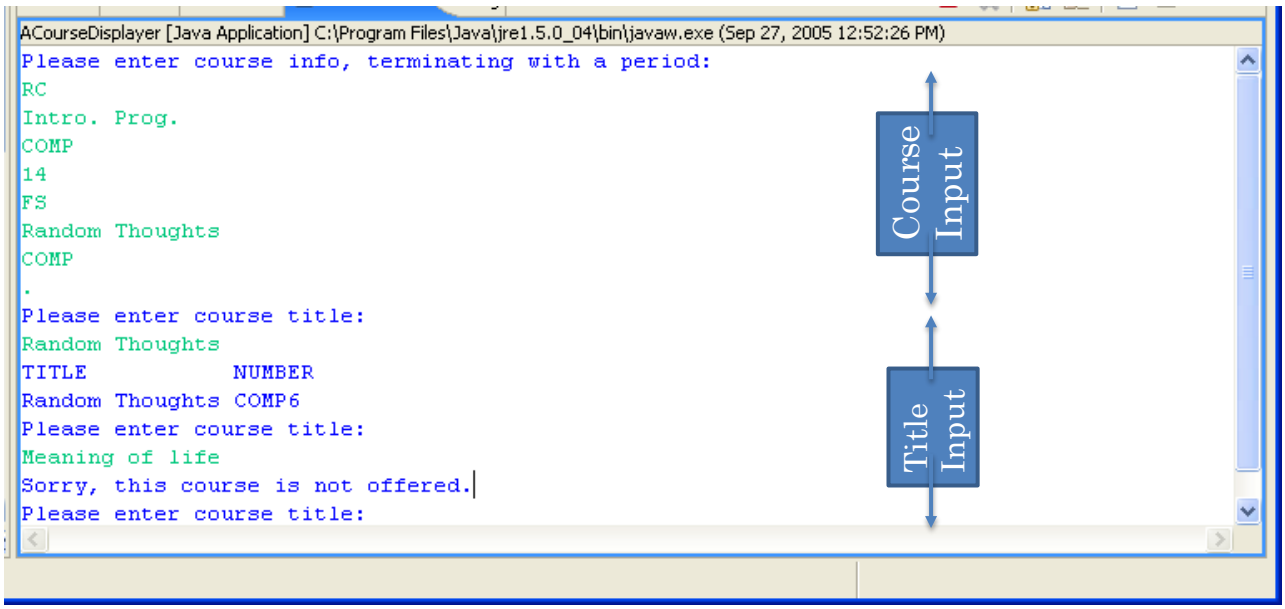


Figure 2 Filling Courses Interactively

The form or syntax of the first part is much more complicated than the second sequence (Figure 27).The second part is simply a series of titles, where each title is a string such as “Random Thoughts.” The first part is similarly a series of course descriptions – the reason why it is more complicated is that the syntax of regular courses and freshman seminars is different, as shown in Figure 27. A regular course description consists of the string “RC,” followed by strings for the title and department, which in turn is followed by the course number. A freshman seminar description is similar except that its starts with “FS” instead of “RC” and does not contain the course number. In figure 27, symbols enclosed within angle brackets are non-terminals and the others are terminals. Terminals are actual strings entered by the user such as “Random Thoughts,” “RC” and “FS”. They are usually tokens returned by a scanner. Non-terminals describe legal sequences of terminals.

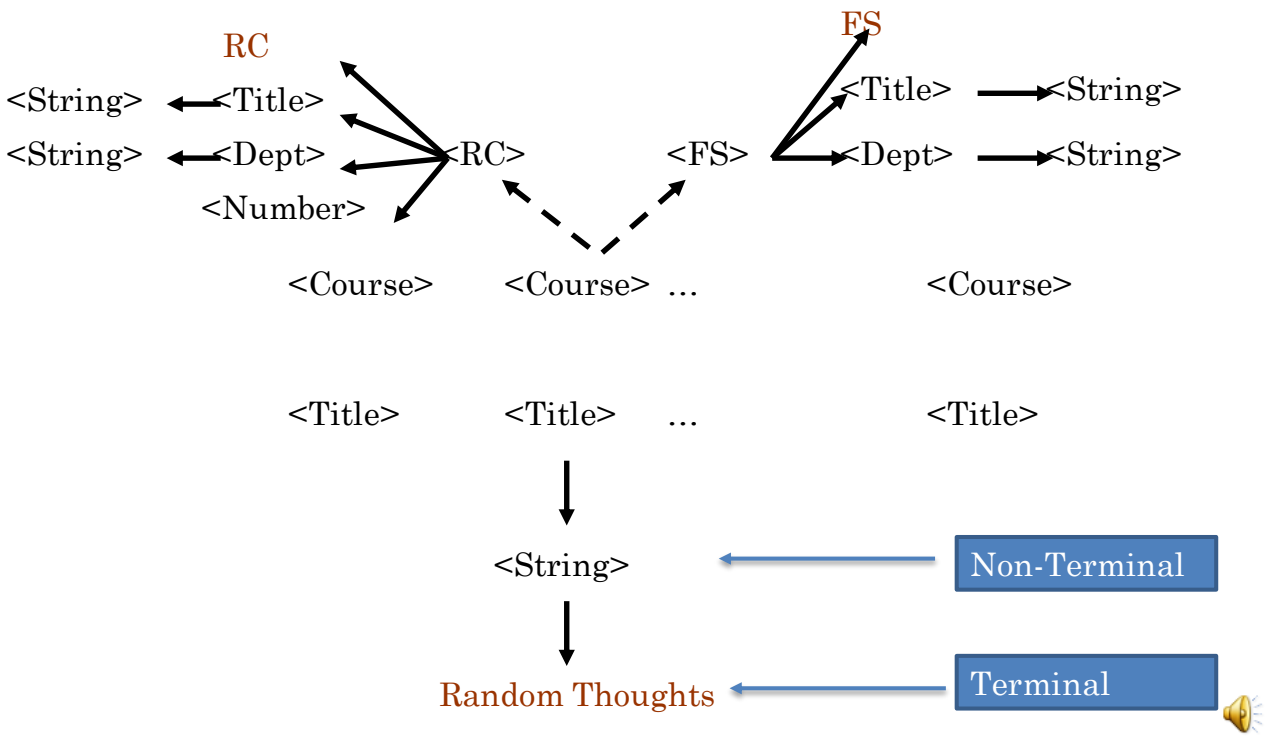


Figure 3 Syntax of input

A more concise description of the syntax is given by the *grammar* given below:

<CourseList> → <Course> *
<Course> → <RC> | <FS>
<RC> → RC <Title> <Dept> <Number>
<FS> → FS <Title> <Dept>
<Title> → <String>
<Dept> → <String>

A grammar consists of a series of rules, where each rule describes a non-terminal on the left hand side in terms of non-terminals and terminals on the right hand side. The symbol "*" says denotes 0 to infinite repetitions of the symbol on its left. There may be more than one rule describing the same non-terminal:

<Course> → <RC>

<Course> → <FS>

Each of these rules describes an alternate syntax for the non-terminal. All of these alternatives can be put in one rule, using the "|" operator:

<Course> → <RC> | <FS>

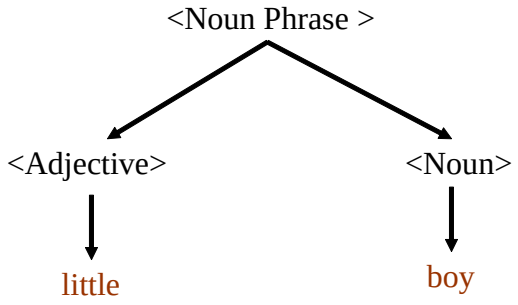
Given more than one syntax for a non-terminal, which one should we choose when processing input? This is similar to the problem we face in writing scanners, given more than one kind of token, which should be try and match? In the case of a scanner, we look at the next character, and see which token it implies. For example, if the next character is a digit, then we try and match a number. We will do something similar when choosing between alternate rules in a grammar. We will determine the first token each of the alternatives requires. As long as the first tokens of different alternatives are different, based on user input, we can determine which rule applies. In the above grammar, based on whether the next token is the terminal "RC" or "FS", we will use the rule <Course> → <RC> or <Course> → <FS>, respectively.

Grammars can be used to describe (unambiguous parts of) a natural language. For example, the following rule describes the syntax of a noun phrase:

<Noun Phrase> → <Adjective> <Noun>
<Adjective> → little | clever | ….
<Noun> → boy | girl | …

T

he following figure shows the use of these rules in the derivation of a legal phrase.



A grammar is associated with the following concepts:

- *Root*: A distinguished non-terminal in the grammar such as `<Course List>` and `<Noun Phrase>`
- *Sentence*: Any sequence of terminals of the grammar.
    - E.g.: `boy little`, `little boy`
- *Legal sentence*: A sequence of terminals that can be derived from the root of the grammar.
    - E.g.: little boy

- *Legal phrase*: A sequence of terminals that can be derived from some non-terminal of the grammar.
    - *E*.g. little
- *Language*: The (possibly infinite) set of legal sentences
    - E.g. {`little boy`, `big boy`, `little girl`, …}
- *Parser*: A program that recognizes a language (set of legal sentences), that is, given a sentence, it can determine if it belongs to the language. Typically, it returns an object representing the sentence, if the sentence belongs to the language.

The process of deriving a sequence of terminals from a non-terminal consists of the following steps:

1. Find a rule in which the non-terminal is the LHS.
2. Replace the non-terminal with the RHS.
3. If the RHS has no non-terminals, then we are done.
4. Otherwise, derive terminals from each of the non-terminals in the RHS.

Thus, to derive “little boy” from <Noun Phrase> we replace it with:

<Adjective> <Noun>

As both of them are non-terminals, we derive terminals from each of them. We first, replace <Adjective> with little to lead to:

little <Noun>

Next we can replace <Noun> with boy, to lead to:

little boy

Now that we understand the grammar of the input of the new user interface, we can start implementing it. Our main method changes very little, instead of calling the original method, `fillCourses()`, which entered hardwired courses into the list, we call `fillCoursesInteractively()`:

```
public class ACourseDisplayer   {

        public static void main(String[] args)      {
                fillCoursesInteractively();
                while (true) {
                        System.out.println("Please enter course title:");
```

```
                        String inputLine = readString();
                        if (inputLine.equals("."))
                                break;
                        Course matchedCourse = courses.matchTitle(inputLine);
                        if (matchedCourse == null)
                                System.out.println("Sorry, this course is not offered.");
                        else {
                                printHeader();
                                print (matchedCourse);
                        }
                }
        }
     static void fillCoursesInteractively() {
                System.out.println("Please enter course info, terminating with a period:");
                courses = (new ACourseParser()).parseCourseList();
    }
}
```

The new method uses a special parser class to build the list. A parser is like a scanner in that it checks if user-input follows the expected syntax and can return an object that represents the input. A parser is a higher-level module that processes a sequence of tokens, while a token processes a sequence of characters. The syntax checked by a parser is usually described by a grammar, as illustrated above. In fact, as we see below, there is a correspondence between the methods in the parser and the rules in the grammar. Before each of these methods, the corresponding grammar rule is given.

Consider first the method, `parseCourseList()` in the parser, which is the one called by `fillCoursesInteractively()`:

```
    //<CourseList> → <Course>*

  public CourseList parseCourseList() {
    CourseList courseList = new ACourseList();
    String nextToken;
    while (true) {
          nextToken = readString();
          if (nextToken.equals("."))
                  break;
          else
                courseList.addElement(parseCourse(nextToken));
    }
    return courseList;
  }
```

It processes a sequence of course descriptions ending with a period, assuming the method `parseCourse()` will process each course description. If the token it reads is not a period, it must be the first token of the course description. Therefore, it passes this token to the `parseCourse()`.

The method `parseCourse()` checks this token. If it is the terminal "FS", it calls `parseFreshmanSeminar()` to process the course description of a freshman seminar. If it is the terminal "RC" then it calls `parseRegularCourse()` to process the course description of a regular course. If it is neither of these terminals, then the input has a parsing error. The method indicates this to the caller by returning the **null** value. When we study exceptions later, we will see a systematic way of propagating error conditions to callers.

```
// <Course> → <RC> | <FS>
Course parseCourse (String firstToken) {
        if (firstToken.toUpperCase().equals("FS"))
                return parseFreshmanSeminar();
        else if (firstToken.toUpperCase().equals("RC"))
                return parseRegularCourse();
        else {
                return null;
        }
}
```

When `parseFreshmanSeminar()` is called, the terminal "FS" has already been removed from the input stream (by `parseCourseList()`). Therefore, all it has to do is read the two strings corresponding to the title and department, and return an instance of AFreshmanSeminar class, passing the two strings as parameters of the constructor of the class:

```
//<FS>  → FS  <Title> <Dept>
Course parseFreshmanSeminar () {
        String title = readString();
        String dept = readString();
        return new AFreshmanSeminar(title, dept);

}
```

`parseRegularCourse()` is very similar. It reads an additional integer for the course number, creates an instance of ARegularCourse(), and passes the two strings and the integer to the constructor of the class.

```
//<RC>  → RC  <Title> <Dept> <Number>
Course parseRegularCourse () {
        String title = readString();
        String dept = readString();
        int number = readInt();
```

```
        return new ARegularCourse (title, dept, number);
}
```

We have seen earlier how to read strings and integers from standard input – so we will not repeat the code for these routines.

The code for our parser of user input is very closely linked to the grammar describing the syntax of the input:

- Each production in the grammar is associated with a parser method. The comments preceding the methods above identify the associated productions.
- The parser method returns an object associated with the LHS of the associated production if a return value expected. For instance, the parseRegularCourse() method, associated with the production, <RC> → RC <Title> <Dept> <Number>, returns an object that represents a regular course. Similarly, the parseCourse() method, associated with the production, <Course> → <FS> | <RC>, also returns an object that can be either a freshman seminar or a regular course. In this case, the interface for a general course is the same as the interface for a regular course.
- A parser method for some production rule can call parser methods for non-terminals in the production, thereby "recursively descending" into lower levels of the tree deriving the input. For instance, parseCourse(), calls parseRegularCourse() to parse the non terminal <RC> in the production, <Course> → <FS> | <RC>.
- The first token in the unparsed input is consumed and used to choose between alternatives. For instance, when parseCourse() is called, it reads (consumed) the next token, and based on its value, chooses between parseRegularCourse() and parseFreshmanSeminar().

A parser with these properties is called a recursive descent parser. It illustrates top-down programming, where we gradually descend into next-level details of a problem by calling methods dealing with these details. For instance, the method for parseCourse() does not have to know what follows RC in a regular course – that is the job of parseRegualrCourse().

Dynamic-Dispatched/Virtual Methods

Consider the following two implementations of a string history.

```
public class AStringHistory implements StringHistory {
        public final int MAX_SIZE = 50;
        String[] contents = new String[MAX_SIZE];
        int size = 0;
        public int size() { return size;}
        public String elementAt (int index) {  return contents[index]; }
        boolean isFull() { return  size == MAX_SIZE; }
        public void addElement(String element) {
           if (isFull())
                   System.out.println("Adding item to a full history");
           else {
                   contents[size] = element;
                   size++;
             }
      }
}
```

```
public class ABigStringHistory implements StringHistory {
        public final int MAX_SIZE = 500;
        String[] contents = new String[MAX_SIZE];
        int size = 0;
        public int size() { return size;}
        public String elementAt (int index) {  return contents[index]; }
        boolean isFull() { return  size == MAX_SIZE; }
        public void addElement(String element) {
           if (isFull())
                   System.out.println("Adding item to a full history");
           else {
                   contents[size] = element;
                   size++;
             }
      }
}
```

The two classes are identical, except for the size of the value of the MAX_SIZE constant! Can we create an abstract superclass that encapsulates some or all of the common code?

Intuitively, it seems we can transfer all of the code in the classes into the superclass and declare only the constant in each subclass. However, this approach does not work as the constant declarations in the subclass are not visible in the superclass. An alternative is to not make the size

of the array a constant – it could be a variable passed to the constructor of the superclass by each subclass. This is a practical alternative that does work. However, it does change the nature of the solution, and for argument's sake, let us say this change is not acceptable. Thus, we need a different alternative.

If the constants are going to be declared in the subclasses, then so must the variable contents, the method elementAt() and almost all of the code in the two classes. The only code that does not need to be declared in the subclasses is the code that checks if a collection is full and prints a message in case this condition is true. However, this code accesses the isFull() method, which the superclass cannot access. One trick we can use is to define a null implementation of this method in the superclass, so it is visible in the class, and then override this method in the subclasses. In fact we can define two such null methods, as the solution below shows:

```
public abstract  class ACollection {
        boolean isFull(){};
        void uncheckedAddElement(Object element){};
        public void addElement(Object element) {
                if (!isFull())
                        uncheckedAddElement(element);
                else
                        handleError();
        }
        void handleError () {
                System.out.println("Adding item to a full history");
        };
}
```

```
public class AStringHistory implements StringHistory {
        public final int MAX_SIZE = 50;
        String[] contents = new String[MAX_SIZE];
        int size = 0;
        public int size() { return size;}
        public String elementAt (int index) {  return contents[index]; }
        boolean isFull() { return size == MAX_SIZE; }
        void uncheckedAddElement(String element) {
                contents[size] = element;
                size++;
          }
  }
```

```
public class ABigStringHistory implements StringHistory {
        public final int MAX_SIZE = 500;
        String[] contents = new String[MAX_SIZE];
        int size = 0;
        public int size() { return size;}
        public String elementAt (int index) {  return contents[index]; }
        boolean isFull() { return size == MAX_SIZE; }
        void uncheckedAddElement(String element) {
                contents[size] = element;
                size++;
          }
    }
```

These two subclasses override the two null methods. Neither class has to worry about checking if the array is full or reporting an error message – this task is handled in the shared class, ACollectiion.

Does this solution really work? The isFull() and uncheckedAddElement() methods are called in the addElement() method of the abstract superclass, ACollection. When this class is written/compiled, the subclasses do not exist. Can addElement() call the overriding methods of the subclasses?

If a called method is statically determined or *dispatched* at compile time, then the answer is no. On the other hand, if the method is dynamically dispatched at execution time, then the answer is yes. C++ supported both kinds of dispatching, and referenced methods statically and dynamically dispatched as regular and virtual methods, respectively. Java (and Smalltalk) support only virtual or dynamically-dispatched methods.

Thus, the call in the superclass does indeed result in execution of the overriding methods, and our solution works.

Abstract Method

However, if the programmers forgets to override the isFull() and unchechedAddElement() methods in the subclasses, the program will compile but not work. One way to reduce this problem is to provide a comment asking the subclass developer to override the methods:

```
// dummy method, please override in each subclass

boolean isFull() { }
```

As the developer may still forget to override, Java provides a solution desired to overcome this problem. An abstract class can declare one or more abstract methods. The difference between an abstract and regular method is that the keyword **abstract** appears in the declaration of an abstract method and the method, like an interface method, has no body. Java ensures that all concrete (direct or indirect) concrete subclasses of the abstract class implement its abstract methods. An abstract class need not implement the abstract methods declared in its (direct or indirect) abstract superclasses.

Thus we can rewrite our abstract class with abstract methods instead of null methods:

```
public abstract class ACollection {
        abstract boolean isFull();
        abstract void uncheckedAddElement(Object element)
        public void addElement(Object element) {
                if (!isFull())
                        uncheckedAddElement(element);
                else
                        handleError();
        }
        void handleError () {
                System.out.println("Adding item to a full history");
        };
}
```

Our concrete classes are identical to what we saw before. However, if these classes omit implementation of either of the two abstract methods, Java will give a compile error.

The following variation of the abstract class ACourse shows the use of an abstract method:

```
package courses;
public abstract class ACourse  {
        String title, dept;
        public ACourse (String theTitle, String theDept) {
                title = theTitle;
                dept = theDept;
        }
        public String getTitle() {
                return title;
        }
         public String getDepartment() {
                return dept;
        }
        abstract public String getNumber();

}
```

Declaring getNumber() as an abstract method ensures that the subclasses of this class, ARegularCourse and AFreshmanSeminar implement this method.

As we see here, with abstract methods, abstract classes can take the place of interfaces. In fact, an abstract class with only abstract methods is essentially an interface.

Abstract Method vs. Interfaces

Thus, to ensure that a class implements a method, we can either declare the method abstract in a (direct or indirect)abstract superclass of the class, or an interface implemented by the class. Which approach should you take?

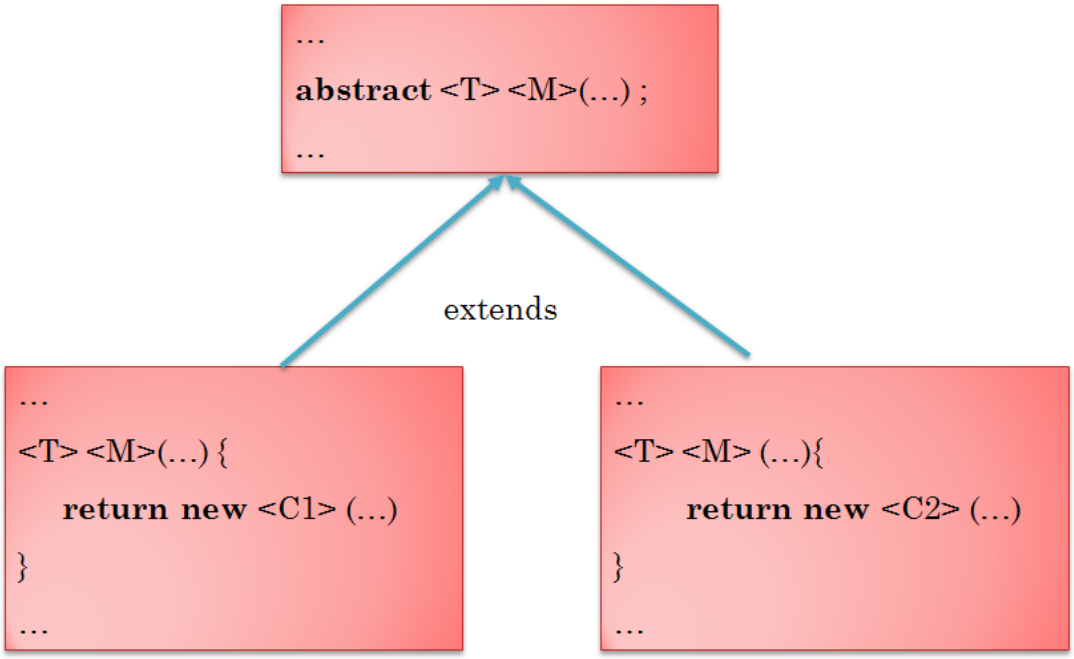
The interface approach is superior, for at least two reasons:

- An interface only contains method headers, while an abstract class can contain both headers and implementations. Thus, it is harder to find the method headers in an abstract class, and thus the specifications of the subclasses of the abstract classes.
- In Java, a class cannot (directly) be a subclass of more than one abstract class whereas it can (directly) implement multiple interfaces. Thus, interfaces provide more flexibility, at least in Java.
- It does not seem right to say a class extends its specification – it implements its specification!

So why does Java support abstract methods? Partly for compatibility with earlier languages such as C++ and Smalltalk that provided them (but did not support interfaces) and partly because abstract methods are more flexible than interface methods in one respect: they do not have to be public. This allows an abstract class to require some implementation-specific method , `a`, to be abstract, and then provide a concrete method, `c`, that uses `a` to provide some common functionality for all subclasses of the class. The isFull() and uncheckedAddElement() methods in the collection examples were examples of such methods. The example was somewhat artificial as we insisted on the max size to be a constant. However, the abstract class we created would be practical if its subclasses used different kinds of objects to store the strings – for instance arrays, array lists, vectors, and linked lists. As we have seen only arrays so far, we put the constant assumption to motivate the abstract class.

Factory Methods

A particularly interesting example of an abstract method is a factory method. This is a method that is expected to return an instance of some type T– hence the name factory. An implementation of the method in a concrete class can instantiate a class (that IS-A T) to create a new instance, or simply return a stored instance. In case a new instance is created, the abstract method can take parameters passed to the constructor used for instantiation. The following figure shows the typical nature of a factory method.



Here the angle brackets around T and M indicate that they can be arbitrary type and method names, respectively, and the … for parameters indicates they can be arbitrary sequences of parameters. In this eample, we assume the concrete methods in the two subclasses return new instances of the expected type, though, as mentioned before, they can return stored instances.

Overriden Methods Accessing Uninitialized Variables

Abstract methods, and overridden method in general, can lead to subtle errors. Consider the following variation of ACourse:

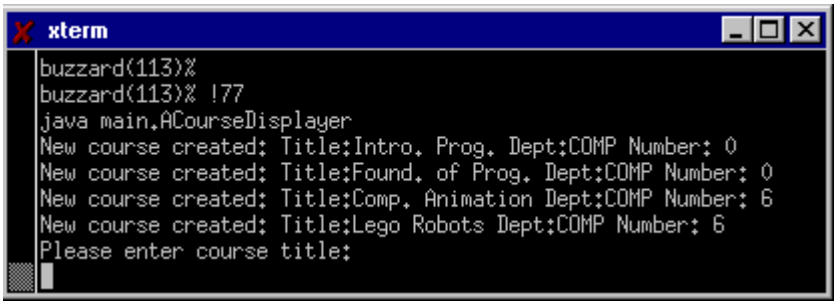
```
package courses;
public abstract class ACourse  {
        String title, dept;
        public ACourse (String theTitle, String theDept) {
                title = theTitle;
                dept = theDept;
                // “innocuous” debugging statement added to ACourse
                System.out.println("New course created: " + "Title:" + title + " Dept:"+ dept + " Number: " +
getNumber())    ;
        }
        public String getTitle() {
                return title;
        }
       public String getDepartment() {
                return dept;
        }
```

```
        abstract public String getNumber();

}
```

The intent there is to print a debugging statement when a course is constructed that displays the title, dept, and number of the course. Rather than changing two constructors in the two concrete classes, respectively, we change a single constructor of the abstract class. Even though the class does not declare a variable or constant for the course number, it can call the abstract method getNumber(), whose two concrete implementations return values that access the variable and constant, respectively.

However, if we run our program, we get the following output:



The correct value of the title and department is printed in all cases. However, for a regular course, the value 0 is printed for the course number rather than the value passed to the constructor. The reason is that the constructor for ACourse calls getNumber() before the constructor for ARegularCourse had had a chance to initialize the number variable. Thus, beware of abstract methods being called in constructors – they may access uninitialized values of variables in subclasses!

A value may be initialized, of course, by an initializing declaration or a constructor. What if we had used an initializing declaration for courseNum in ARegularCourse:

```
public class ARegularCourse extends ACourse implements Course {
        int courseNum = 99;
        public ARegularCourse (String theTitle, String theDept, int theCourseNum) {
                super (theTitle, theDept);
                courseNum = theCourseNum;
        }
        public int getNumber() {
                return courseNum;
        }
}
```

As it turns out, the program still displays the default value of the course number. The reason is that the constructor of a superclass is called before the initializations in the declarations of the subclass are processed.

Let us take the example of the new statement given below to understand what happens:

```
courses.addElement(new ARegularCourse ("Intro. Prog.",  "COMP", 14));
```

- The variables declared in ARegularCourse and Course are allocated space but not initialized.
- The constructor of ARegularCourse is called.
- It calls the constructor of ACourse in its first statement.
- The initializations in the declarations of the variables of ACourse are processed. In this case there no initializations.
- The constructor of ACourse is started.
- The two instance variables of ACourse are assigned parameter values, “Intro. Prog.” And “COMP”.
- The values of these variables are printed.
- The method getNumber() of ARegularCourse is called and the default value of courseNum is printed.
- The constructor of ACourse returns.
- The initializations in the declarations of the variables of ARegularCourse are processed. In this case, courseNum is assigned the value 0.
- Execution resumes in the constructor of ARegularCourse.
- The courseNum variable is assigned the parameter value, 14.
- The constructor of ARegularCourse returns.
- The new statement completes, returning the new instance to its caller.

If the initializations in the declarations of the subclass are processed after the superclass constructor returns, then why did the correct value of SEMINAR_NUMBER get printed above? The reason is that a constant is initialized when it is allocated. The Java compiler creates a single memory location for all instances of the constant, which is initialized when the class declaring it is loaded in memory. Thus, unlike a variable, it is not initialized when an instance of the class is created.

Access Control on Variables

Consider now the following change to ARegularCourse:

```
package courses;

public class ARegularCourse extends ACourse implements Course {

        int courseNum = 99;

        public ARegularCourse (String theTitle, String theDept, int theCourseNum) {

                …

                title = theTitle;

        }

        public int getNumber() {

                return courseNum;

        }

}
```

Here the constructor of ARegularCourse accesses the variable, title, declared in the superclass, ACourse.

Even though title is not a public variable, this is, in fact, legal, because both classes are in the same package. In general, when declaring a member (variable, constant, method) in some class, we can give it one of four access types, with decreasing restriction:

- public: accessible in all classes.
- protected: accessible in all subclasses of its class and all classes in its package.
- default: accessible in all classes in its package.
- private: accessible only in its class.

We give default access by not specifying any access modifier; we give some other access by specifying it as a modifier, as in the following two declarations:

```
protected String title;
private String title;
```

Had we declared title as private, we would not have been able to access it in the subclass.

Default access also allows AFreshmanSeminar to set the title:

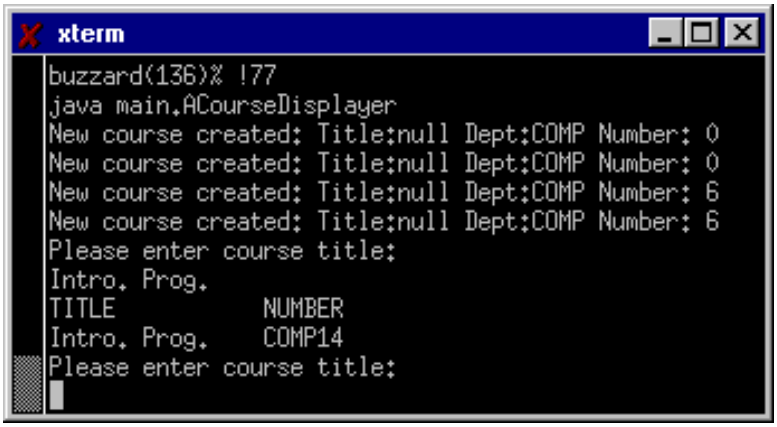
```
package courses;
public class AFreshmanSeminar extends ACourse implements FreshmanSeminar {
        public AFreshmanSeminar (String theTitle, String theDept) {
                super (theTitle, theDept);
                title = theTitle;
        }
        public int getNumber() {
                return SEMINAR_NUMBER;
        }
}
```

Now it is no longer necessary to set title in ACourse:

```
package courses;
public abstract class ACourse  {
        String title, dept;
        public ACourse (String theTitle, String theDept) {
                //title = theTitle;
                dept = theDept;
                // “innocuous” debugging statement added to ACourse
                System.out.println("New course created: " + "Title:" + title + " Dept:"+ dept + " Number: " +
getNumber())    ;
        }
        public String getTitle() {
                return title;
         }
       public String getDepartment() {
                return dept;
         }
```

```
        abstract public String getNumber();
}
```

As it turns out, these three changes do change the output of the program, as shown below:



This time, the constructor prints uninitialized values of not only course numbers of regular courses but also the title of all courses. The reason is that this variable is initialized in the constructors of the subclasses now, which have not performed the initialization when this output is generated. As we can see, the program continues to match titles correctly, as the getTitle() method is called after all constructors have executed and hence returns the correct value.

This, of course, not the correct way to program this example as initialization of the variable is being duplicated in the two classes. However, we may turn up with this code if we first created the two concrete classes as independent classes without a common superclass and then factored the common code into a common superclass. We may have forgotten to move the initialization code to the superclass.

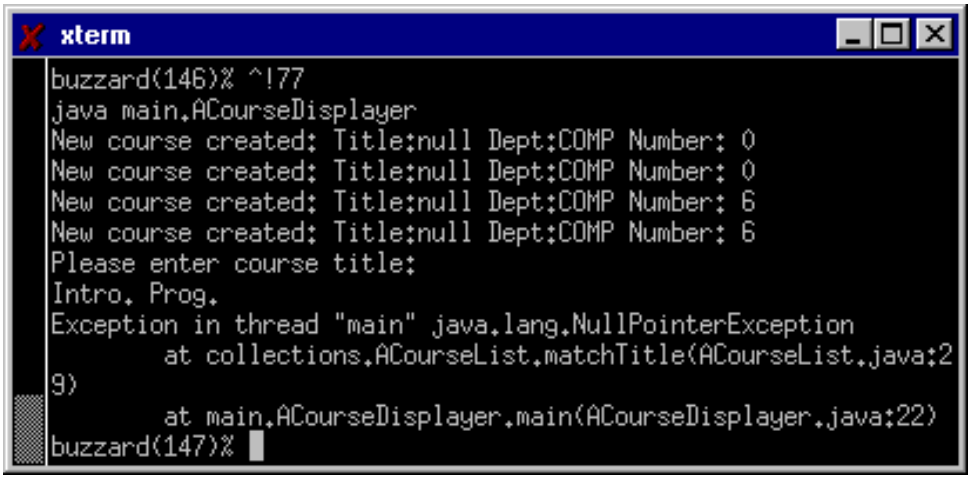
Redeclaring a variable in a subclass

In fact, we might even forget to delete one or more of the common variables moved to the superclass from the subclasses:

```
package courses;
public class AFreshmanSeminar extends ACourse implements FreshmanSeminar {
                String title;
                …
}

public class ARegularCourse extends ACourse implements Course {
        String title;
}
```

Surprisingly, Java will not catch this error. It allows redeclaration of superclass variables, which some other languages, such as Smalltalk, did not allowed. Perhaps the reason is to be consistent – if redeclaration of methods is allowed, why not variables also? However, this policy can lead to runtime errors such as the one shown below.



The reason for the exception is the following: the initialization of the title, done in the two subclasses, set the value of the copy of the variable declared in the subclasses. The getTitle() method called to match the input string, however, referenced the copy in the superclass, which never got initialized. The null pointer exception is thrown when matchTitle() tries to call the equals() method on the value returned by getTitle().

It was non-trivial to come up with this reasoning even when we knew about the error – it would have been extremely hard to do so if we did not know about the error and almost impossible if we were not familiar with Java's rule of allowing re-declarations. Thus, beware of inadvertent re-declarations!

Mulitple Constructors

A class can have multiple constructors, as shown in yet another variation of ACourse below:

```
package courses;
public abstract class ACourse  {
        final String DEFAULT_DEPT = "COMP";    final String DEFAULT_TITLE = "Topics in Computer
Science";
        String dept, title;          String dept = "COMP";
        public ACourse (String theTitle, String theDept) {
                title = theTitle;
```

```
                dept = theDept;
                System.out.println("New course created: " + "Title:" + title + " Dept:"+ dept + " Number: " +
getNumber())    ;
    }
        public ACourse () {
                dept = DEFAULT_DEPT;
                title = DEFAULT_TITLE;
                System.out.println("New course created: " + "Title:" + title + " Dept:"+ dept + " Number: "
+ getNumber()) ;
        }
        public String getTitle() {return title;}
        public String getDepartment() {return dept};
        abstract public String getNumber();
}
```

Here, we have added a parameterless constructor that simples uses default values of dept and title. The definitions of the subclasses of this class do not call this constructor, so let us change AFreshmanSeminar to do so:

```
package courses;
public class AFreshmanSeminar extends ACourse implements FreshmanSeminar {
        public AFreshmanSeminar (String theTitle, String theDept) {
                super (theTitle, theDept);
        }
        public AFreshmanSeminar () {}
        public int getNumber() {
                return SEMINAR_NUMBER;
        }
}
```

Here we have added a second parameterless constructor that does nothing. Recall that if a constructor does not call a superclass constructor, then Java automatically puts a call to the parameterless superclass constructor. Thus, the above constructor definition is equal to:

```
        public AFreshmanSeminar () {super();}
```

Java would have added this constructor automatically had we defined no other constructor. Since we have defined a two-parameter constructor, we must add this definitional manually.

Either constructor can now be used to create an instance of this class:

new AFreshmanSeminar (“Lego Robots”, “COMP”);
new AFreshmanSeminar();

Init method

Notice that the two constructors of the abstract class duplicate code. A straightforward solution to this problem is define the common code in a separate method:

```
package courses;
public abstract class ACourse  {
        final String DEFAULT_DEPT = “COMP”;  final String DEFAULT_TITLE = “Topics in Computer Science”;
        String dept, title;    public ACourse (String theTitle, String theDept) {init (theTitle, theDept);}
        public ACourse () {  init (DEFAULT_TITLE, DEFAULT_DEPT );}
        void init (String theTitle, String theDept) {
                title = theTitle;
                dept = theDept;
                System.out.println("New course created: " + "Title:" + title + " Dept:"+ dept + " Number: " +
getNumber());
        }
        public String getTitle() {return title;}
        public String getDepartment() {return dept};
        abstract public String getNumber();
}
```

We will call a non-constructor method such as init() above that initializes an object as an init method. In fact, it is standard practice to call it init().

A more subtle solution to this problem is to make the parameterless constructor directly call the parameterized constructor:

```
package courses;
public abstract class ACourse  {
        final String DEFAULT_TITLE = “Topics in Computer Science”;
        final String DEFAULT_DEPT = “COMP”;
        String title, dept;
        public ACourse (String theTitle, String theDept) {
                title = theTitle;
                dept = theDept;
                System.out.println("New course created: " + "Title:" + title + " Dept:"+ dept + " Number: "
+ getNumber()) ;
        }
        public ACourse () {
                 this (DEFAULT_TITLE, DEFAULT_DEPT);
        }
```

```
 public String getTitle() {return title;}
 public String getDepartment() {return dept};
 abstract public String getNumber();
}
```

As it turns out, the Java implementations I have seen so far will complain. Some will simply say that a constructor cannot call another constructor in the same class, while others will be more specific and say that the problem is that the parameters to the constructor, DEFAULT_TITLE and DEFAULT_DEPT, may not have been initialized. If we try passing literals as parameters:

```
public ACourse () {
        this (“Topics in Computer Science”, “COMP”;);
}
```

the implementations I have seen will allow the call, even those that said constructors cannot call other constrictors.

So what is going on? In general, calling a constructor from another constructor in the same class is a confusing concept as we must relate it to invocation of superclass constructors. Recall that our rule so far was that a constructor (implicitly or explicitly) calls a superclass constructor. This means that for each of the constructors calls in a class, a superclass constructor is called. In the above example, when ACourse() is called, super() is called, and then when ACourse (String theTitle, String theDept) is called, super () is called again. Thus, the superclass would be initialized multiple times. If each initialization created the same result, then this would not be a problem, but there is no way to guarantee this – recall there may be different constructors in a superclass doing different things and the same constructor may do different things each time it is called or may have side effects.

A solution is to call the superclass constructor only in the last constructor call in a chain of constructor calls. Thus, in the example above, when ACourse() is called, super() is not called, but when ACourse (String theTitle, String theDept) is called, since it is the last call in this two-cal chain, super() is called. This is the approach taken in Java. However, a danger here is that the parameters to a constructor call may not have correct initial values as the superclass has not been initialized. For example, if we had the following definition of ACourse():

```
public ACourse () {
        this (title, dept);
}
```

then the danger is that title and dept may depend on some variables declared in the superclass, which have not yet been initialized as yet as no superclass constructor has been called. As a result the above code will be rejected by Java.

However, rejecting our original code:

```
public ACourse () {
        this (DEFAULT_TITLE, DEFAULT_DEPT);
}
```

makes no sense, as Java knows at this point that these are initialized final variables whose values cannot change after the superclass initialization has occurred! The only reason is that the compiler writers were too lazy to distinguish between initialized final variables and other kinds of variables!

Now consider the following alternative approach to reusing the code:

```
package courses;
public abstract class ACourse  {
        String tite = “Topics in Computer Science”;;
        String dept = “COMP”;
        public ACourse (String theTitle, String theDept) {
                title = theTitle;
                dept = theDept;
                this();
        }
        public ACourse () {
                System.out.println("New course created: " + "Title:" + title + " Dept:"+ dept + " Number: " +
getNumber());
        }
        public String getTitle() {return title;}
        public String getDepartment() {return dept};
        abstract public String getNumber();
}
```

Here we have used initializing declarations to set the default values of title and dept. We have put the print statement now in the default constructor, which is called by the parameterized constructor after initializing the values of the two variables.

As it turns out, this is illegal too, as a constructor call, like a super call, must be the first statement of in a constructor. This danger is that code before the constructor call may reference variables in the superclass before the superclass constructor is invoked by the last constructor in the call chain.

By writing init() methods, we don’t have to worry about the unreasonable handling of constructors in Java implementations. They are highly recommended for at least two other reasons. First, they allow

initialization after an object is created. This can be useful when the creator and initializer are different objects. As we might see later, special factories may be used to create objects of various kinds, which may not know how to initialize them, leaving this task to their clients (the factory users). Finally, unlike constructors, init methods are class independent and thus can be put in interfaces. This makes it possible to initialize, with one piece of code, objects of multiple classes that implement a common interface.

To illustrate the separation of object creation and initialization, consider yet another variation of ACourse:

```
package courses;
public abstract class ACourse   {
        String title = “Topics in Computer Science”;;
        String dept = “COMP”;
        public ACourse (String theTitle, String theDept) {
                init(theTitle, theDept);
        }
         public void init (String theTitle, String theDept) {
                title = theTitle;
                dept = theDept;
        }
        public ACourse () { }
        public String getTitle() {return title;}
        public String getDepartment() {return dept};
        abstract public String getNumber();
}
```

This time, we rely on initializing declarations to assign default values to the instance variables. The null constructor, thus, does nothing. The init method takes the same arguments as the parameterized constructor, which simply calls init(). This time it has been declared public so that a user of the object can call it after creating the object using the unparameterized constructor. Its header should now go into the interface Course.

We can follow the same approach in changing ARegularCourse:

```
public class ARegularCourse extends ACourse implements Course {
        int courseNum = 99;
         public ARegularCourse (String theTitle, String theDept, int theCourseNum) {
                init (theTitle, theDept, theCourseNum);
        }
        public ARegularCourse() {
        }
        public  void init (String theTitle, String theDept, int theCourseNum) {
                courseNum = theCourseNum;
        }
```

```
        public int getNumber() {
                return courseNum;
        }
}
```

Again, the init method takes the same argument as the parameterized constructor and is called by the latter. However, this code does not have the intended effect. The original parameterized constructor:

```
public ARegularCourse (String theTitle, String theDept, int theCourseNum) {
                super (theTitle, theDept);
                courseNum = theCourseNum;
}
```

called the parameterized superclass constructor, while the new implementation does not. Since there is no explicit call to super, the unparameterized constructor is called, which is not what we want. Moreover, if init() is called after the object is created using the unparamaterized constructor, then the superclass init is not called. To solve the second problem, we must have each init call an appropriate init in the superclass:

```
public  void init (String theTitle, String theDept, int theCourseNum) {
                init(theTitle, theDept);
                courseNum = theCourseNum;
}
```

This also solves the first problem. When the parameterized constructor is called, Java sees that init() is the first call, and notices that init() calls super() before accessing any variables in the class. It views this call to super() as safe. Again, the header of the method should go into the interface, Course.

We can now construct and initialize the object together in a single call to the parameterized constructor:

```
new ARegularCourse (“Intro. Prog”, “COMP”, 14);
```

or we can first create the object using the unparameterized constructor and then initialized it using init():

```
(new ARegularCourse()).init(“Intro Prog.”, “COMP”, 14)
```

The second approach is dangerous as you may forget to initialize an object after constructing it. You will see its use when we study factories.

In the examples above, we created a single parameterized constructor that initialized all instance variables. It One can imagine creating multiple parameterized constructors depending on how may default values are overridden by a constructor. For example, we can add the following to ACourse:

```
public ACourse (String theTitle) {
        title = theTitle;
}
```

This constructor uses the default value of dept (assigned by the initializing declaration), but overrides the default value of title.

In general, you should write an init method for each parameterized constructor.

Polymorphic vs. overloaded vs. dynamically dispatched methods

Invocations of three kinds of methods take arguments of different types and are therefore often confused with each other:

- Overloaded methods: As we see before, an overloaded method is a method that shares its name with other methods that take parameters of different types. Which one of these methods sharing a method name is called is determined at compile time based on the types of the arguments provided in the call.
- Polymorphic methods: This is a method that takes arguments of multiple types.
- Dynamically dispatched methods: Like an overloaded method, this is a method that shares its name with other methods. However, it differs from an overloaded method in three different ways. First, it is constrained to be an instance method while an overloaded method can be an instance or class method. Second, the choice of the called method is determined, not on basis of the argument type, but the target object. As it is an instance method, a target object is guaranteed. Finally, the choice is made dynamically at execution time rather than at compile time. C++ calls such a method as a virtual method.

We have seen above examples of all three kinds of methods. The following code shows all three kinds:

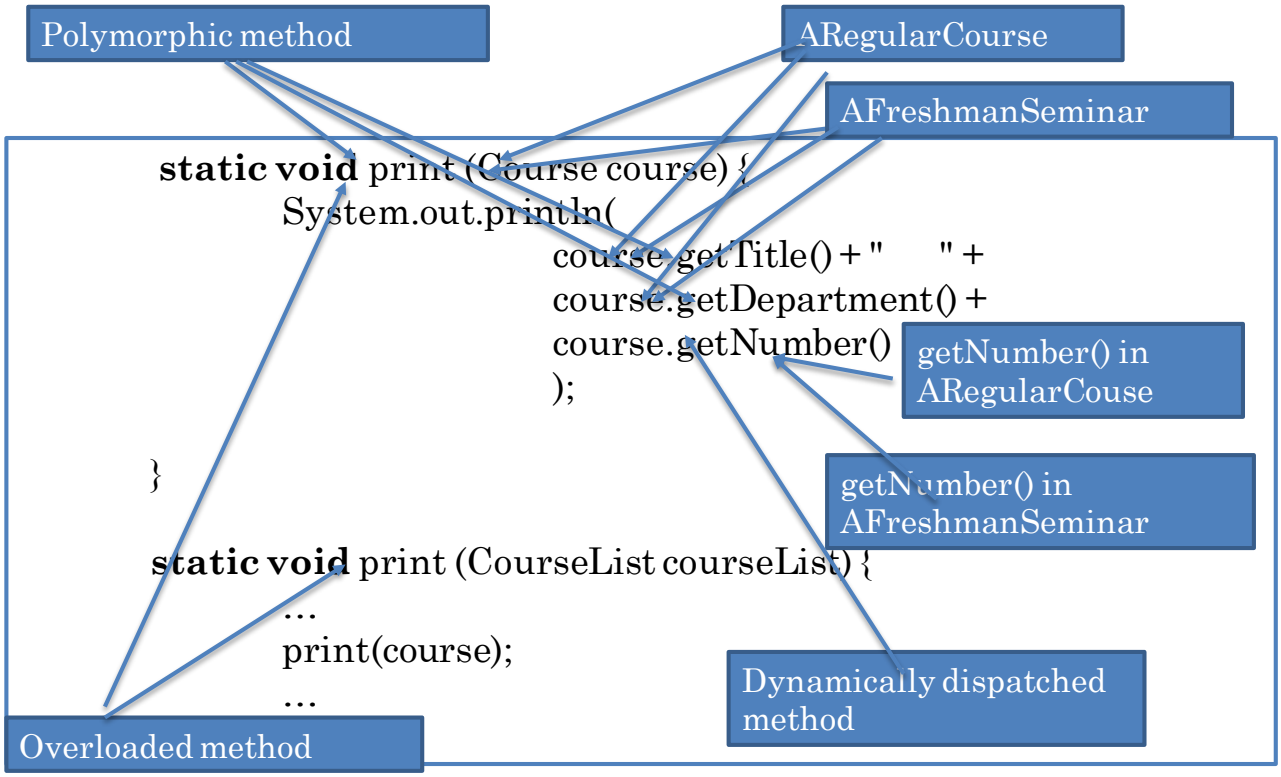


Figure 4 Overloaded vs. Polymorphic vs. dynamically dispatched method

print() is an overloaded method because two definitions of it exist above, one taking an argument of type Course and another of type CourseList. Each of these definitions is also a polymorphic method as it can take arguments of different types. For example, the first print can be applied to an instance of ARegularCourse or AFreshmanSeminar. Finally, the getNumber() method invoked above is a dynamically dispatched method. Two definitions of it exist – one in AFreshmanSeminar and another in ARegularCourse. Which one is selected depends on the class of the actual object stored in the target variable, course – it is the one provided by that class. Thus, if the print method is passed an instance of AFreshmanSeminar, then the getNumber() method provided by this class is used.

Older languages such as C and Pascal do not allow programmers to write such methods though they do support predefined overloaded operators such as +, which works on integers and real numbers. What are the benefits of supporting them?

The most obvious benefit is that they allow us to remember fewer method names. For example, one needs to only remember that print is the name of a method that prints a value of any type to the screen, rather than remembering separate names for different types such as printPoint and printDouble. This makes both program writing and understanding easier.

A more subtle benefit is that they prevent programmers from having to manually dispatch methods, that is, test the type of a value and call different methods based on the test. The following code shows a program performing a manual dispatch::

```
static void print (Course course) {

int number;

if (course instanceof ARegularCourse)

        number = ((ARegularCourse)course).getNumberFreshmanSeminar();

else if (course instanceof AFreshmanSeminar)

        number = ((AFreshmanSeminar)course).getNumberRegularCourse();

System.out.println(

                course.getTitle() + "          " +

                course.getDepartment() +

                number

                );

}
```

The **instanceof** operator used above checks if the object on the left is an instance of the type (class/interface) specified on the right. An object is an instance of its class, the superclasses of its class, the interfaces implemented by its class, and the supertypes of these interfaces. In the above example, this operator is used to determine the class of the course object passed to be printed, and based on the test, the object is cast to the appropriate class and a method specific to the class is called. Manual dispatch require additional code to test the type of an object. Worse, this code must be updated whenever the set of types of the object being tested increases. For example, if we were to add a new course, ASeniorSeminar, the test used above would have to be updated. By using a class-independent name for the two method, putting this name in a common interface implemented by the class, and using the dynamic dispatch capability of java we have been able to avoid manual dispatch in this example. It is not always possible to avoid manual dispatch, as in the following example:

```
static void print (RegularCourse course) {
        int number;
        if (course instanceof RegularCourse)
                System.out.print(“Regular Course: ”);
        else if (course instanceof AFreshmanSeminar)
                System.out.println(“Freshman Seminar”);
```

```
        System.out.println(
                        course.getTitle() + "       " +
                        course.getDepartment()
                        );
        }
```

It is not possible to create a polymorphic method that handles the two cases because different things must be done in each case. We might try to use overloading to fix this problem:

```
static void print (Course course) {
        printHeader (course);
        System.out.println(
        course.getTitle() + "       " +
        course.getDepartment()  +
        course.getNumbert() );
}
static void printHeader (ARegularCourse course) {
        System.out.print("Regular Course: ");
}
static void printHeader (AFreshmanSeminar course) {
        System.out.print("Freshman Seminar: ");
}
```

However, recall that Java chooses between overloaded methods at compile time. When the compiler processes:

```
        printHeader (course);
```

it has no idea, which of the two printHeader() methods should be called, as instances of both course classes can be assigned to `course`. In other wordfs, it is not possible to create two overloaded methods in an external class as overload resolution occurs at compile time, and in this example, at compile time it is not possible to say which of the two cases will be executed.

It is possible to create two different dynamically dispatched methods that print the headers:

```
package courses;
public class ARegularCourse extends ACourse implements Course {
        …
        public void printHeader () {
            System.out.print ("Regular Course: ")
       }
}
package courses;
```

```
public class AFreshmanSeminar extends ACourse implements FreshmanSeminar {
        …
        public void printHeader () {
            System.out.print (“Freshman Seminar: ”)
       }
}
```

Now we can indeed write a polymorphic print method that does not use instanceof:

```
static void print (Course course) {
    course.printHeader();
    System.out.println(
        course.getTitle() + "       " +
        course.getDepartment()  +
        course.getNumbert() );
}
```

However, this solution is undesirable as one as user interface code such as printing a header must not mix with computation code.

Thus, we must use **instanceof**!

Similarly, an expression parser may expect an instance of a unary minus operator followed by an instance of a number

-3

or simply an instance of a number:

3

In the two cases, it needs to do different thing. Thus, it would have to do an instanceof check on the token to determine what it actually got.

For the reasons given above, wherever possible, try using a polymorphic/overloaded/dynamically dispatched method.

A polymorphic method offers the additional benefit of reuse of code: a single piece of code is used to process values of different types. In overloaded and dynamically dispatched methods, different method implementations are used for different types. Therefore, use polymorphism over the two the other two techniques by, wherever possible.

- Using interfaces over classes as types.
- Using supertypes over subtypes as types.

It is not always possible to use polymorphic methods over the other two kinds of methods. For example, the getCourseNumber() method above cannot be polymorphic as the algorithm to calculate the course number depends on the class of the object. Similarly, the two print() methods above (print (Course) and print (CourseList)) cannot be polymorphic as they must do different things based on whether a course or a course list is being printed. The first print method (print(Course)), on the other hand, could have been replaced by two overloaded definitions:

```
static void print (AFreshmanCourse course) {
                System.out.println(
                                course.getTitle() + "      " +
                                course.getDepartment() +
                                course.getNumber()
                                );

}
static void print (ARegularCourse course) {
                System.out.println(
                                course.getTitle() + "      " +
                                course.getDepartment() +
                                course.getNumber()
                                );

}
```

Similarly, the polymorphic getDepartment() method defined in the abstract class ACourse could have been replaced by two dynamically dispatched methods in the two concrete classes. When overloaded/dymanically dispatched methods do the same thing, combine them into a polymorphic method, which may involve creation of additional interfaces/super types.

How about dynamic dispatch vs. overloading? The answer depends on several factors. Recall that a dynamically dispatched method is declared in the class of the object on which the method works, that is, it is the target of the method, while an overloaded method is declared in an external class and is provided the object as a parameter. Thus, the answer depends on whether the method must be in the class of the object on which it operates or an external class, which is a program decomposition issue. For example, user-interface methods such as the print would be overloaded while non user-interface methods such as getCourseNumber would be dynamically dispatched. Sometimes these are not real alternatives as overloading is more general in that it looks at the types of multiple parameters while dynamic dispatch looks only at the type of the single target object. Overloading may be chosen because of this increased generality. On the other hand, dynamic dispatch may be chosen because it is done at runtime while overloading is resolved at compile time.

It is not possible to simply look at method calls to determine if they refer to overloaded, polymorphic, or dynamically dispatched methods. It is not possible to determine if they refer to polymprhic or dynamically dispatched methods. For instance, the call to getDepartment() looks exactly like the call to getNumber() above. However, getDepartment() is a polymorphic method as a single definition of it exists in the abstract class ACourse. Similarly, it is not possible to determine if they refer to polymorphic or overloaded methods. For example, based on the following calls:

```
print (new AFreshmanSeminar(…));

print (new ARegualrCourse(…));
```

it is not possible to say whether these two calls refer to one polymorphic method or two overloaded methods.

Summary

- Deleting an element of an ordered collection involves moving a two-element window along the array, assigning the second element of the window to the first one..
- Java allows classes and interfaces to inherit declarations in existing classes and methods, adding only the definitions needed to extend the latter.
- An inherited method can be overridden by a new method.
- Inheritance and implementation are examples of IS-A relationships.
- If T2 IS-A T1, then a value of type T2 can be assigned to a variable of type T1.
- A class can have multiple constructors.
- A class should have an unparameterized constructor and one or more init() methods to allow code reuse and initialization after object creation.
- Polymorphic, overloaded and dynamically dispatched methods allow reuse of method names and relieve programmers from manually testing the type of an object.
- Polymorphic methods are preferred to overloaded and dynamically dispatched methods as they support reuse of the same code for values of different types.
- Whether a method is overloaded or dynamically dispatched depends on various factors.

1. A user-interface method would be overloaded while a computation method would be dynamically dispatched.
2. When multiple types are needed to map a call to an implementation, overloading would be preferred.
3. When the mapping must be done at runtime, dynamic dispatch would be chosen.

- Sometimes neither polymoprhism, nor overload resolution, nor dynamic dispatch work and manual testing of object types must be done.
- A grammar consists of rules involving terminals and non-terminals that describe the syntax of user input.
- Terminals are tokens entered by the user, while non-terminal describe sequences of legal tokens.
- A non-terminal may be associated with more than one rule – the first token of each alternative can be used to decide which rule to use.
- A module that enforces the syntax described by a grammar is called a parser. It processes tokens produced by a scanner.
- In a well-written parser, each grammar rule is associated with a corresponding method.
- The first token of the rule enforced by a method may have been consumed by its caller to decide which alternative to use.