COMP 401

DYNAMIC DISPATCH AND VIRTUAL AND ABSTRACT METHODS

Instructor: Prasun Dewan
Prerequisites

- Inheritance
- Abstract Classes
public class APointHistory implements PointHistory {
    public final int MAX_SIZE = 50;
    protected Point[] contents = new Point[MAX_SIZE];
    protected int size = 0;
    public int size() {
        return size;
    }
    public Point elementAt (int index) {
        return contents[index];
    }
    protected boolean isFull() {return size == MAX_SIZE;}
    public void addElement(int x, int y) {
        if (isFull())
            System.out.println("Adding item to a full history");
        else {
            Point p = new ACartesianPoint(x, y);
            contents[size] = p;
            size++;
        }
    }
}
public class APolarPointHistory implements PointHistory {
    public final int MAX_SIZE = 50;
    protected Point[] contents = new Point[MAX_SIZE];
    protected int size = 0;
    public int size() {
        return size;
    }
    public Point elementAt (int index) {
        return contents[index];
    }
    protected boolean isFull() { return size == MAX_SIZE; }
    public void addElement(int x, int y) {
        if (isFull())
            System.out.println("Adding item to a full history");
        else {
            Point p = new APolarPoint(x, y);
            contents[size] = p;
            size++;
        }
    }
}
A PolarPointHistory Implementation

```java
public class APolarPointHistoryWithAddMethod extends APointHistory {
    public void addElement(int x, int y) {
        if (isFull())
            System.out.println("Adding item to a full history");
        else {
            Point p = new APolarPoint(x, y);
            contents[size] = p;
            size++;
        }
    }
}
```

Only line changed, still code duplication
public class ACartesianPointHistoryWithDynamicallyDispatchedMethod
    implements PointHistory {
    public final int MAX_SIZE = 50;
    protected Point[] contents = new Point[MAX_SIZE];
    protected int size = 0;
    public int size() {
        return size;
    }
    public Point elementAt (int index) {
        return contents[index];
    }
    protected boolean isFull() {
        return size == MAX_SIZE;
    }
    public void addElement(int x, int y) {
        if (isFull())
            System.out.println("Adding item to a full history");
        else {
            Point p = createPoint(x, y);
            contents[size] = p;
            size++;
        }
    }
    protected Point createPoint(int x, int y) {
        return new ACartesianPoint(x, y);
    }
}
public class APolarPointHistoryWithDynamicallyDispatchedMethod extends ACartesianPointHistoryWithDynamicallyDispatchedMethod {
    protected Point createPoint(int x, int y) {
        return new ACartesianPoint(x, y);
    }
}

public static void main (String[] args) {
    PointHistory pointHistory = new APolarPointHistoryWithDynamicallyDispatchedMethod();
    pointHistory.addElement(50, 100);
}
public class ACartesianPointHistoryWithDynamicallyDispatchedMethod implements PointHistory {

    public class APolarPointHistoryWithDynamicallyDispatchedMethod extends ACartesianPointHistoryWithDynamicallyDispatchedMethod {
        protected Point createPoint(int x, int y) {
            return new ACartesianPoint(x, y);
        }
    }

    private PointHistory pointHistory = new APolarPointHistoryWithDynamicallyDispatchedMethod();
    pointHistory.addElement(50, 100);

    public void addElement(int x, int y) {
        if (isFull())
            System.out.println("Adding item to a full history");
        else {
            Point p = createPoint(x, y);
            contents[size] = p;
            size++;
        }
    }

    protected Point createPoint(int x, int y) {
        return new ACartesianPoint(x, y);
    }
}

Static dispatching: call most specific method known when caller method was compiled

Dynamic dispatching: call most specific method for the calling object
**Virtual vs. Regular Method and Different Languages**

- **Virtual method:** Call to it is dynamically dispatched

- **Regular method:** Call to it is statically dispatched

- **Java:** All instance methods are virtual methods and all class (static) methods are regular methods

- **C++:** Instance methods can be declared to be regular or virtual and all class (static) methods are regular methods

- **Smalltalk:** All methods are virtual
INHERITANCE RELATIONSHIPS?

APolarPointHistoryWithDynamicallyDispatchedMethod

IS-A

ACartesianPointHistoryWithDynamicallyDispatchedMethod

ACartesianPointHistoryWithDynamicallyDispatchedMethod

IS-A

APolarPointHistoryWithDynamicallyDispatchedMethod
**Correct Inheritance**

- AnAbstractPointHistory
  - IS-A A CartesianPointHistory
  - IS-A A PolarPointHistory
CORRECT INHERITANCE

AnAbstractPointHistory

IS-A

ACartesianPointHistory

IS-A

APolarPointHistory
public class ACartesianPointHistoryWithDynamicallyDispatchedMethod implements PointHistory {

    public final int MAX_SIZE = 50;
    protected Point[] contents = new Point[MAX_SIZE];
    protected int size = 0;
    public int size() {
        return size;
    }
    public Point elementAt (int index) {
        return contents[index];
    }
    protected boolean isFull() {
        return size == MAX_SIZE;
    }
    public void addElement(int x, int y) {
        if (isFull())
            System.out.println("Adding item to a full history");
        else {
            Point p = createPoint(x, y);
            contents[size] = p;
            size++;
        }
    }
    protected Point createPoint(int x, int y) {
        return new ACartesianPoint(x, y);
    }
}
public abstract class AnAbstractPointHistory implements PointHistory {
    public final int MAX_SIZE = 50;
    protected Point[] contents = new Point[MAX_SIZE];
    protected int size = 0;
    public int size() {
        return size;
    }
    public Point elementAt (int index) {
        return contents[index];
    }
    protected boolean isFull() {
        return size == MAX_SIZE;
    }
    public void addElement(int x, int y) {
        if (isFull())
            System.out.println("Adding item to a full history");
        else {
            Point p = createPoint(x, y);
            contents[size] = p;
            size++;
        }
    }
}
public abstract class AnAbstractPointHistory implements PointHistory {
    public final int MAX_SIZE = 50;
    protected Point[] contents = new Point[MAX_SIZE];
    protected int size = 0;
    public int size() {
        return size;
    }
    public Point elementAt (int index) {
        return contents[index];
    }
    protected boolean isFull() {
        return size == MAX_SIZE;
    }
    public void addElement(int x, int y) {
        if (isFull())
            System.out.println("Adding item to a full history");
        else {
            Point p = createPoint(x, y);
            contents[size] = p;
            size++;
        }
    }
    protected Point createPoint(int x, int y) {}
}
public class ACartesianPointHistory extends AnAbstractPointHistory{
    protected Point createPoint(int x, int y) {
        return new ACartesianPoint(x, y);
    }
}
Erroneous Implementation with No Errors

```java
public class ACartesianPointHistory extends AnAbstractPointHistory{
}
```
public abstract class AnAbstractPointHistory implements PointHistory {
    public final int MAX_SIZE = 50;
    protected Point[] contents = new Point[MAX_SIZE];
    protected int size = 0;
    public int size() {
        return size;
    }
    public Point elementAt (int index) {
        return contents[index];
    }
    protected boolean isFull() {
        return size == MAX_SIZE;
    }
    public void addElement(int x, int y) {
        if (isFull())
            System.out.println("Adding item to a full history");
        else {
            Point p = createPoint(x, y);
            contents[size] = p;
            size++;
        }
    }
}

protected abstract Point createPoint(int x, int y);
Java complains abstract method not implemented in concrete class
Null vs. Abstract Methods

Null methods could replace abstract methods.

Abstract method force overriding implementations in subclasses.

Provide better documentation, indicating to programmer that subclass will implement them.
ABSTRACT METHOD

- Declared only in abstract classes
- Keyword: `abstract`
- No body
- Each (direct or indirect) subclass must implement abstract methods defined by an abstract class.
- Much like each class must implement the methods defined by its interface(s).
ABSTRACT CLASS VS. METHODS

- Abstract method => containing class abstract
  - Cannot have an unimplemented method in an instance
- Abstract class may not contain abstract method
A Course with Abstract Method

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

    public String toString() {
        return "Title:" + title + " Dept:" + dept + " Number: " + getNumber();
    }
}
```

Abstract Method
An abstract class can implement any interface

This means all sub classes implement the interface

Interface implementation check is made for concrete classes

**EQUIVALENT WITH INTERFACE**

```java
public abstract class ACourse implements Course {
    String title, dept;
    public ACourse (String theTitle, String theDept) {
        super();
        title = theTitle;
        dept = theDept;
    }
    public String getTitle() {
        return title;
    }
    public String getDepartment() {
        return dept;
    }
    public String toString() {
        return "Title:" + title + " Dept:" + dept + " Number:" + getNumber();
    }
}
```
package courses;
public abstract class ACourse implements Course {
    String title, dept;
    public ACourse (String theTitle, String theDept) {
        super();
        title = theTitle;
        dept = theDept;
    }
    public String getTitle() {
        return title;
    }
    public String getDepartment() {
        return dept;
    }
    public String toString() {
        return "Title:" + title + " Dept:" + dept + " Number: " +
    }
}

An abstract class can implement any interface
This means all sub classes implement the interface
Interface implementation check is made for concrete classes
public class ARegularCourse extends ACourse {
    int courseNum;
    public ARegularCourse (String theTitle, String theDept, int theCourseNum) {
        super (theTitle, theDept);
        courseNum = theCourseNum;
    }
    public int getNumber() {
        return courseNum;
    }
}

Saying ARegularCourse implements Course is redundant
public class AFreshmanSeminar extends ACourse {
    public AFreshmanSeminar (String theTitle, String theDept) {
        super (theTitle, theDept);
    }
    public int getNumber() {
        return SEMINAR_NUMBER;
    }
}
public abstract class AnAbstractPointHistory implements PointHistory {
    public final int MAX_SIZE = 50;
    protected Point[] contents = new Point[MAX_SIZE];
    protected int size = 0;
    public int size() {
        return size;
    }
    public Point elementAt (int index) {
        return contents[index];
    }
    protected boolean isFull() {
        return size == MAX_SIZE;
    }
    public void addElement(int x, int y) {
        if (isFull())
            System.out.println("Adding item to a full history");
        else {
            Point p = createPoint(x, y);
            contents[size] = p;
            size++;
        }
    }
    protected abstract Point createPoint(int x, int y);
}
Adding `createPoint` to Interface

```java
public interface PointHistoryWithExtraPublicMethod {
    public Point createPoint(int x, int y);
}
```

```java
public abstract class AnAbstractPointHistoryWithExtraPublicMethod implements PointHistoryWithExtraPublicMethod {
    public int MAX_SIZE = 50;
    private Point[] contents = new Point[MAX_SIZE];
    private int size = 0;

    public int size() {
        return size;
    }

    public Point elementAt(int index) {
        return contents[index];
    }

    protected boolean isFull() {
        return size == MAX_SIZE;
    }

    public void addElement(int x, int y) {
        if (isFull())
            System.out.println("Adding item to a full history");
        else {
            Point p = createPoint(x, y);
            contents[size] = p;
            size++;
        }
    }
}
```

Factory method

Concrete class guaranteed to implement it

Least privilege violated
Factory Abstract Method

```
...  
abstract <T> <M>(...) ;
...

...  
<T> <M>(...) {
    return new <C1> (...)
}
...

...  
extends

...  
<T> <M>(...){
    return new <C2> (...)
}
...
```

Abstract method that returns an instance of some type T – like a factory, it creates and initializes an object.
Abstract methods vs. Interfaces

- Unimplemented methods become abstract methods to be implemented by concrete subclasses. Do not need explicit implements clause in subclass if no additional public methods implemented,
  - public class ARegularCourse extends ACourse {
    ....
  }

- A class with only abstract methods simulates interfaces.
- No multiple inheritance in Java
- Separation of abstract and concrete methods with interfaces.
- A class implements but does not inherit a specification!
ABSTRACT METHODS VS. INTERFACES

- Non-public abstract methods relevant even in Java
- All interface methods must be public.
- May want non public abstract methods. E.g.
  - `abstract boolean isFull();`
Abstract methods must be virtual methods.

Virtual methods need not be abstract methods.
**FACTORY METHODS VS. ABSTRACT METHODS**

- Some abstract methods may be factory methods
- Factory methods may not be abstract methods but often this means bad programming practice

Diagram:

```
APolarPointHistoryWithDynamicallyDispatchedMethod

IS-A

ACartesianPointHistoryWithDynamicallyDispatchedMethod
```
Overloading, Polymorphism, and Dynamic dispatch

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

static void print (CourseList courseList) {
    ...
    print(course);
    ...
}

Polymorphic method

ARegularCourse

AFreshmanSeminar

getNumber() in ARegularCouse

dynamically dispatched method

getNumber() in AFreshmanSeminar

Overloaded method

Polymorphic method

Dynamically dispatched method

Polymorphic method
Overloading, Polymorphism, and Dynamic Dispatch

- Same method name works on different types.

  - **Overloading**
    - Multiple (static or instance) methods with same name but different parameter types.
    - Resolved at compile time based on parameter types.

  - **Polymorphism**
    - Same method implementation works on multiple object types.

  - **Dynamic dispatch (or virtual methods)**
    - Multiple instance methods with same name in different classes
    - Resolved at execution time based on type of target object
Polymorphism vs. Overloading and Dynamic Dispatch

- **Polymorphism vs. Overloading**
  - Polymorphism: single `print (Course course)`
  - Overloading: `print (ARegularCourse course)` and `print (AFreshmanSeminar course)` with same implementation.

- **Polymorphism vs. Dynamic Dispatch**
  - Polymorphism: single `getTitle()` in `ACourse`
  - Dynamic dispatch: `getTitle()` in `AFreshmanSeminar()` and `getTitle()` in `ARegularCourse()` with same implementation.

- **Create polymorphic code when you can**
  - In overloading and dynamic dispatch, multiple implementations associated with each method name.
  - In polymorphic case, single implementation.
    - Use interfaces rather than classes as types of parameters
    - Use supertypes rather than subtypes as types of parameters
POLYMORPHISM VS. OVERLOADING AND DYNAMIC DISPATCH

- Cannot always create polymorphic method.
  - `getNumber()` for `ARegularCourse` and `AFreshmanSeminar` do different things.
  - `print(Course course)` and `print (CourseList courseList)` do different things.

- When polymorphism not possible try overloading and dynamic dispatch.
**MANUAL INSTEAD OF DYNAMIC DISPATCH**

```java
static void print (Course course) {
    if (course instanceof ARegularCourse)
        number = ((ARegularCourse)course).
                    getCourseNumberRegularCourse();
    else if (course instanceof AFreshmanSeminar)
        number = ((AFreshmanSeminar)course).
                    getCourseNumberFreshmanSeminar();

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

More Work

Must update code as subtypes added
CAN WE GET RID OF MANUAL DISPATCH/INSTANCEOF?

```java
static void print (Course course) {
    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() +
        course.getNumber() );
}
```
CANNOT ALWAYS GET RID OF MANUAL DISPATCH/INSTANCOF

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

At compile time, do not know if regular course or freshman seminar

Actual parameter cannot be supertype of formal parameter
package courses;
public class ARegularCourse extends ACourse implements Course {

    ...  

    public void printHeader () {
        System.out.print ("Regular Course: ")
    }
}

PRINTHEADER() IN AREGULARCOURSE
package courses;
public class AFreshmanSeminar extends ACourse implements FreshmanSeminar {

    ... 

    public void printHeader () {
        System.out.print ("Freshman Seminar: ")
    }
}
Should not always get rid of manual dispatch/instanc eof

```java
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) {
}

Should not put printHeader() in ARegularCourse or AFreshmanSeminar as it is UI code.

Hence need to use instanceof

Used in parsers
```