Problem Solving

Creating a class from scratch

Recipe for Writing a Class

1. Write the class boilerplate stuff
2. Declare Fields
3. Write Creator(s)
4. Write accessor methods
5. Write mutator methods
6. Write test cases / test driver
7. Now go back and fill in the implementation

Class Boilerplate

- What is the class name?
- What package will the class be in?
- Javadoc Description

```
package xmpPackage;
/*
 * Keep track of examples
 */
class XmpClass {
}
```

Boilerplate:
the parts of your code that is always the same and doesn't change the function but is still needed

Declare Fields

- What data needs to be tracked for this class?
- What data type does that need to be?
- As we implement the class, we may modify the list of fields, but we should at least get started.

```
private int xmpFld; // example number
```

Create Constructor(s)

- What parameters are required?
- Which fields need / can have initial values?
- How do we derive initial field values from parameters?
- Write boilerplate… then fill in

```
/*
  * Create an xmpClass object
  * @param number to initialize field
  */
public XmpClass(int initVal) {
}
```

Create getter accessor methods

- This is just typing… really mindless
- For each field (at least each one you will need outside the class)
  - Method name is get<field_name>
  - Return type is the type of the field
  - Return value is the field name

```
/*
  * @returns current value of xmpFld
  */
public int getXmpFld() { return xmpFld; }
```

Create non-getter accessor methods

- What parameters do you get?
- What value do you need to return? What type is it?
- How do you get from fields and parameters to return value?
- Write boilerplate first

```
/*
  * @returns double field value
  */
public int getDoubleFld() {
}
```

Accessor method “toString”

- If you want to print your object, think about how you want it formatted.
- When your object is “appended” to a string, your “toString” method will be invoked
- If you don’t specify a toString method, a default toString method will be used… it may be good enough. If not… write your own.

```
public String toString() { return “xmpFld=” +xmpFld; }
```

Mutator methods - setters

- Again, mindless
- Write only if the outside world directly needs to modify fields
- For each set field:
  - Returns void
  - Name is set<field_name>
  - Argument is the same type as the field

```
/*
  * @param new value for field
  */
public void setXmpFld(int n) { xmpFld=n; }
```

Mutator methods - Actions

- What action is being performed on the object?
- What parameters are required to perform that action?
- How should the action modify the fields?
- Does this action return a value? If so, what type of value?

```
/*
  * add to the field value
  * @param value to add
  * @return new value
  */
public int add(int n) {
}
```

Unit Test Driver

- Test all methods at least once
- Test all exception cases
- Read through code… what might go wrong
  - What might be null?
  - What might overflow?
  - What causes divide by zero?
  - What might violate your assumptions?
  - Remember… if it's really exceptional, let exception handling handle it.

In-class vs. External Unit Test Driver

In-Class

- Packaged with the class
- More likely up to date
- Easy to run
- May miss private/public problems

External

- More like how your class will be used
- Catch all public/private problems
- Requires more discipline to keep up to date
- Requires extra management for extra classes

Unit Test Boilerplate

```
public static void main(String[] arg) {
        System.out.println("Test XmpClass creators");
        XmpClass test1 = new XmpClass(14);
        if (!"xmpFld=14".equals(test1.toString())) {
                System.out.println("new XmpClass(14) failed!");
        }
        …
}
```

Problem

Write a Java class that models three boxes that can contain balls called "Boxes". The Boxes class should have a single field – an array of integers that defines how many balls are in each of the three boxes. The creator should take three integer parameters that define the initial state of the boxes – the number of balls in each box. The Boxes class should also have a "move" method, which takes two parameters – an integer 0, 1, or 2 to represent the "from" box, and an integer 0, 1, or 2 to represent the "to" box. The move method should return a Boolean "true" value if the move is successful, that is – it is possible to move a ball from the "from" box to the "to" box. In this case, the move method should update the number of balls in the "from" box and the number of balls in the "to" box. If the move is not possible, the move method should return "false" and not modify the number of balls in the boxes. You may assume the "from" and "to" parameters contain valid values.

First, write the class boilerplate…

```
package quiz1;

public class Boxes {
}
```

Declare Fields

Write a Java class that models three boxes that can contain balls called "Boxes". The Boxes class should have a single field – an array of integers that defines how many balls are in each of the three boxes. The creator should take three integer parameters that define the initial state of the boxes – the number of balls in each box. The Boxes class should also have a "move" method, which takes two parameters – an integer 0, 1, or 2 to represent the "from" box, and an integer 0, 1, or 2 to represent the "to" box. The move method should return a Boolean "true" value if the move is successful, that is – it is possible to move a ball from the "from" box to the "to" box. In this case, the move method should update the number of balls in the "from" box and the number of balls in the "to" box. If the move is not possible, the move method should return "false" and not modify the number of balls in the boxes. You may assume the "from" and "to" parameters contain valid values.

Declare Fields…

```
package quiz1;

public class Boxes {
        int[] balls;
}
```

Write Creator Boilerplate

Write a Java class that models three boxes that can contain balls called "Boxes". The Boxes class should have a single field – an array of integers that defines how many balls are in each of the three boxes. The creator should take three integer parameters that define the initial state of the boxes – the number of balls in each box. The Boxes class should also have a "move" method, which takes two parameters – an integer 0, 1, or 2 to represent the "from" box, and an integer 0, 1, or 2 to represent the "to" box. The move method should return a Boolean "true" value if the move is successful, that is – it is possible to move a ball from the "from" box to the "to" box. In this case, the move method should update the number of balls in the "from" box and the number of balls in the "to" box. If the move is not possible, the move method should return "false" and not modify the number of balls in the boxes. You may assume the "from" and "to" parameters contain valid values.

Creator Boilerplate

```
package quiz1;

public class Boxes {
        int[] balls;
        public Boxes(int p1, int p2, int p3) {
        }
}
```

Create accessor/action method

Write a Java class that models three boxes that can contain balls called "Boxes". The Boxes class should have a single field – an array of integers that defines how many balls are in each of the three boxes. The creator should take three integer parameters that define the initial state of the boxes – the number of balls in each box. The Boxes class should also have a "move" method, which takes two parameters – an integer 0, 1, or 2 to represent the "from" box, and an integer 0, 1, or 2 to represent the "to" box. The move method should return a Boolean "true" value if the move is successful, that is – it is possible to move a ball from the "from" box to the "to" box. In this case, the move method should update the number of balls in the "from" box and the number of balls in the "to" box. If the move is not possible, the move method should return "false" and not modify the number of balls in the boxes. You may assume the "from" and "to" parameters contain valid values.

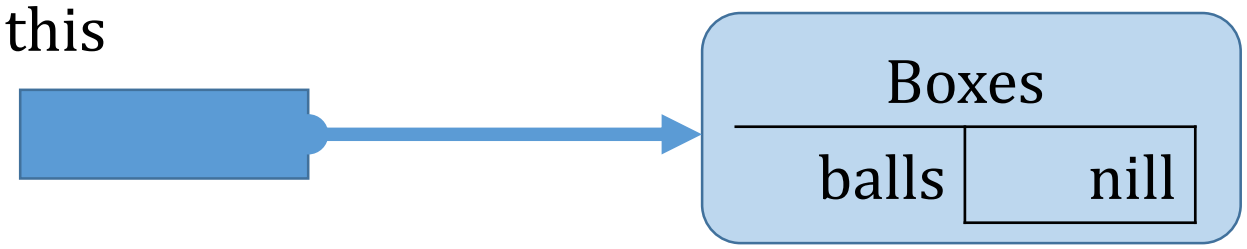
Method Boilerplate

```
package quiz1;

public class Boxes {
	int[] balls;
	public Boxes(int p1, int p2, int p3) {
	}
	boolean move(int from,int to) {
	}
}
```

Implement Creator

Write a Java class that models three boxes that can contain balls called "Boxes". The Boxes class should have a single field – an array of integers that defines how many balls are in each of the three boxes. The creator should take three integer parameters that define the initial state of the boxes – the number of balls in each box. The Boxes class should also have a "move" method, which takes two parameters – an integer 0, 1, or 2 to represent the "from" box, and an integer 0, 1, or 2 to represent the "to" box. The move method should return a Boolean "true" value if the move is successful, that is – it is possible to move a ball from the "from" box to the "to" box. In this case, the move method should update the number of balls in the "from" box and the number of balls in the "to" box. If the move is not possible, the move method should return "false" and not modify the number of balls in the boxes. You may assume the "from" and "to" parameters contain valid values.

At the start of creator …



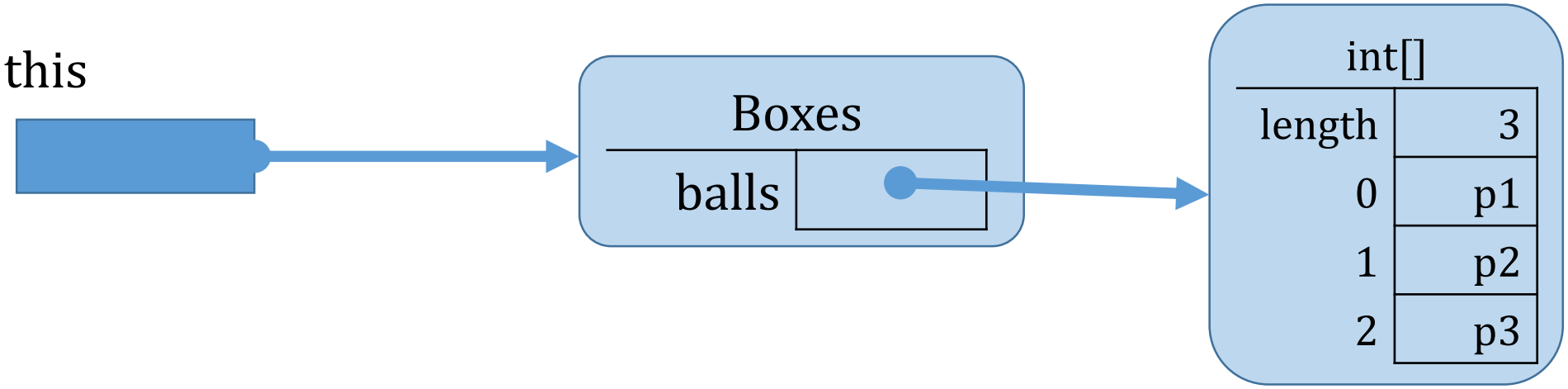
Instantiate the Array!!!!!!

```
package quiz1;

public class Boxes {
        int[] balls;
        public Boxes(int p1, int p2, int p3) {
                balls=new int[3];
                balls[0]=p1; balls[1]=p2; balls[2]=p3;
        }
        …
```

Alternative Instantiate the Array!!!!!!

```
package quiz1;

public class Boxes {
        int[] balls;
        public Boxes(int p1, int p2, int p3) {
                int[] temp={p1,p2,p3};
                balls=temp;
        }
        …
```

At end of creator …



Implement the methods

Write a Java class that models three boxes that can contain balls called "Boxes". The Boxes class should have a single field – an array of integers that defines how many balls are in each of the three boxes. The creator should take three integer parameters that define the initial state of the boxes – the number of balls in each box. The Boxes class should also have a "move" method, which takes two parameters – an integer 0, 1, or 2 to represent the "from" box, and an integer 0, 1, or 2 to represent the "to" box. The move method should return a Boolean "true" value if the move is successful, that is – it is possible to move a ball from the "from" box to the "to" box. In this case, the move method should update the number of balls in the "from" box and the number of balls in the "to" box. If the move is not possible, the move method should return "false" and not modify the number of balls in the boxes. You may assume the "from" and "to" parameters contain valid values.

Implement move

```
public class Boxes {
        int[] balls;
        …
        boolean move(int from,int to) {
                if (balls[from]==0) return false;
                balls[from]--; balls[to]++ return true;
        }
}
```

Implement toString

```
public class Boxes {
        int[] balls;
        …
        public String toString() {
                return “boxes A[“ + balls[0] +”] B[“ + balls[1] +
                        “C[“ + balls[2]  “]”;
        }
}
```

Write Tester

```
package quiz1;

class Tester {
        static public void main(String args[]) {
                Boxes b = new Boxes(3,2,1);
                System.out.println("Boxes start at: " + b);

                while(b.move(0,2)) {}
                System.out.println("After loop, its: " + b);
        }
}
```

Run Test

```
laptop:~/cs140/quiz1>java -cp . quiz1.Tester
Boxes start at: Boxes A[3] B[2] C[1]
After loop, its: Boxes A[0] B[2] C[4]
laptop:~/cs140/quiz1>
```