6.092: Introduction to Java

6: Design, Debugging, Interfaces
Assignment 5: main()

Programs start at a main() method, but many classes can have main()

```java
public class SimpleDraw {
    /* ... stuff ... */

    public static void main(String args[]) {
        SimpleDraw content = new SimpleDraw(new DrawGraphics());
        /* ... more stuff ... */
    }
}
```
Assignment 5: main()

Programs start at a main() method, but many classes can have main()

```java
public class SimpleDraw {
    /* ... stuff ... */

    public static void main(String args[]) {
        SimpleDraw content = new SimpleDraw(new DrawGraphics());
        /* ... more stuff ... */
    }
}
```
public class DrawGraphics {
    BouncingBox box;

    public DrawGraphics() {
        box = new BouncingBox(200, 50, Color.RED);
    }

    public void draw(Graphics surface) {
        surface.drawLine(50, 50, 250, 250);
        box.draw(surface);
    }
}
public class DrawGraphics {
    BoundingBox box; // a field or member variable

    public DrawGraphics() {
        box = new BoundingBox(200, 50, Color.RED);
    }

    public void draw(Graphics surface) {
        surface.drawLine(50, 50, 250, 250);
        box.draw(surface);
    }
}
public class DrawGraphics {
    BouncingBox box;

    public DrawGraphics() { // constructor
        box = new BouncingBox(200, 50, Color.RED);
    }

    public void draw(Graphics surface) {
        surface.drawLine(50, 50, 250, 250);
        box.draw(surface);
    }
}
public class DrawGraphics {
    public void draw(Graphics surface) {
        surface.drawLine(50, 50, 250, 250);
        box.draw(surface);
        surface.fillRect(150, 100, 25, 40);
        surface.fillOval(40, 40, 25, 10);
        surface.setColor(Color.YELLOW);
        surface.drawString("Mr. And Mrs. Smith", 200, 10);
    }
}
public class DrawGraphics {
    ArrayList<BouncingBox> boxes = new ArrayList<BouncingBox>();

    public DrawGraphics() {
        boxes.add(new BouncingBox(200, 50, Color.RED));
        boxes.add(new BouncingBox(10, 10, Color.BLUE));
        boxes.add(new BouncingBox(100, 100, Color.GREEN));
        boxes.get(0).setMovementVector(1, 0);
        boxes.get(1).setMovementVector(-3, -2);
        boxes.get(2).setMovementVector(1, 1);
    }

    public void draw(Graphics surface) {
        for (BouncingBox box : boxes) {
            box.draw(surface);
        }
    }
}
Outline

Good program design

Debugging

Interfaces
What is a good program?

Correct / no errors

Easy to understand

Easy to modify / extend

Good performance (speed)
Consistency

Writing code in a consistent way makes it easier to write and understand

Programming “style” guides: define rules about how to do things

Java has some widely accepted “standard” style guidelines
Naming

Variables: Nouns, lowercase first letter, capitals separating words
x, shape, highScore, fileName

Methods: Verbs, lowercase first letter
getSize(), draw(), drawWithColor()

Classes: Nouns, uppercase first letter
Shape, WebPage, EmailAddress
Good Class Design

Good classes: easy to understand and use

- Make fields and methods private by default
- Only make methods public if you need to
- If you need access to a field, create a method:

```java
public int getBar() { return bar; }
```
Debugging

The process of finding and correcting an error in a program

A fundamental skill in programming
Step 1: Don’t Make Mistakes

Don’t introduce errors in the first place
Step 1: Don’t Make Mistakes

Don’t introduce errors in the first place

• Reuse: find existing code that does what you want
• Design: think before you code
• Best Practices: Recommended procedures/techniques to avoid common problems
Design: Pseudocode

A high-level, understandable description of what a program is supposed to do

Don’t worry about the details, worry about the structure
Pseudocode: Interval Testing

Example:

Is a number within the interval \([x, y)\)?

If number \(< x\) return false
If number \(> y\) return false
Return true
Design

Visual design for objects, or how a program works

Don’t worry about specific notation, just do something that makes sense for you

Scrap paper is useful
SimpleDraw

DrawGraphics

ArrayList

BouncingBox

BouncingBox

BouncingBox
Step 2: Find Mistakes Early

Easier to fix errors the earlier you find them

- Test your design
- Tools: detect potential errors
- Test your implementation
- Check your work: assertions
Testing: Important Inputs

Want to check all “paths” through the program.
Think about one example for each “path”

Example:
Is a number within the interval \([x, y)\)?
Intervals: Important Cases

Below the lower bound
Equal to the lower bound
Within the interval
Equal to the upper bound
Above the upper bound
Intervals: Important Cases

What if lower bound > upper bound?

What if lower bound == upper bound?

(hard to get right!)
Pseudocode: Interval Testing

Is a number within the interval \([x, y)\)?

If number < x return false
If number > y return false
Return true
Pseudocode: Interval Testing

Is a number within the interval \([x, y)\)?

Is 5 in the interval \([3, 5)\)?

If number < x return false
If number > y return false
Return true
Pseudocode: Interval Testing

Is a number within the interval \([x, y)\) ?

Is 5 in the interval \([3, 5)\) ?

If number \(< x\) return false
If number \(\geq y\) return false
Return true
Tools: Eclipse Warnings

Warnings: may not be a mistake, but it likely is.

Suggestion: always fix all warnings

Extra checks: FindBugs and related tools
Unit testing: JUnit makes testing easier
Assertions

Verify that code does what you expect

If true: nothing happens
If false: program crashes with error
Disabled by default (enable with -ea)

```c
assert difference >= 0;
```
```java
void printDifferenceFromFastest(int[] marathonTimes) {
    int fastestTime = findMinimum(marathonTimes);

    for (int time : marathonTimes) {
        int difference = time - fastestTime;
        assert difference >= 0;
        System.out.println("Difference: "+ difference);
    }
}
```
Step 3: Reproduce the Error

• Figure out how to repeat the error
• Create a minimal test case

Go back to a working version, and introduce changes one at a time until the error comes back

Eliminate extra stuff that isn’t used
Step 4: Generate Hypothesis

What is going wrong?
What might be causing the error?

Question your assumptions: “x can’t be possible:” What if it is, due to something else?
Step 5: Collect Information

If x is the problem, how can you verify? Need information about what is going on inside the program.

System.out.println() is very powerful.

Eclipse debugger can help.
Step 6: Examine Data

Examine your data

Is your hypothesis correct?

Fix the error, or generate a new hypothesis
Why Use Methods?

Write and test code once, use it multiple times: avoid duplication

Eg. Library.addBook()
Why Use Methods?

Use it without understanding *how* it works: *encapsulation / information hiding*

Eg. How does `System.out.println()` work?
Why Use Objects?

Objects combine a related set of variables and methods

Provide a simple *interface*

( encapsapsulation again)
Library

Book[] books;
int numBooks;
String address;

void addBook(Book b) {
    books[numBooks] = b;
    numBooks++;
}

Library

void addBook(Book b);
Java Interfaces

Manipulate objects, without knowing how they work

Useful when you have similar but not identical objects

Useful when you want to use code written by others
Interface Example: Drawing

```java
public class BouncingBox {
    public void draw(Graphics surface) {
        // … code to draw the box …
    }
}

// … draw boxes …
for (BouncingBox box : boxes) {
    box.draw(surface);
}
```
Interface Example: Drawing

```java
public class Flower {
    public void draw(Graphics surface) {
        // … code to draw a flower …
    }
}

// … draw flowers …
for (Flower flower : flowers) {
    flower.draw(surface);
}
```
public class DrawGraphics {
    ArrayList<BoundingBox> boxes = new ArrayList<BoundingBox>();
    ArrayList<Flower> flowers = new ArrayList<Flower>();
    ArrayList<Car> cars = new ArrayList<Car>();

    public void draw(Graphics surface) {
        for (BoundingBox box : boxes) {
            box.draw(surface);
        }
        for (Flower flower : flowers) {
            flower.draw(surface);
        }
        for (Car car : cars) {
            car.draw(surface);
        }
    }
}
public class DrawGraphics {
    ArrayList<Drawable> shapes = new ArrayList<Drawable>();
    ArrayList<Flower> flowers = new ArrayList<Flower>();
    ArrayList<Car> cars = new ArrayList<Car>();

    public void draw(Graphics surface) {
        for (Drawable shape : shapes) {
            shape.draw(surface);
        }
        for (Flower flower : flowers) {
            flower.draw(surface);
        }
        for (Car car : cars) {
            car.draw(surface);
        }
    }
}
Interfaces

Set of classes that share methods

Declare an *interface* with the common methods

Can use the interface, without knowing an object’s specific type
Interfaces: Drawable

```java
import java.awt.Graphics;

interface Drawable {
    void draw(Graphics surface);
    void setColor(Color color);
}
```
Implementing Interfaces

Implementations provide complete methods:

```java
import java.awt.Graphics;

class Flower implements Drawable {
    // ... other stuff ...
    public void draw(Graphics surface) {
        // ... code to draw a flower here ...
    }
}
```
Interface Notes

Only have methods (mostly true)

Do not provide code, only the definition (called signatures)

A class can implement any number of interface
Using Interfaces

Can only access stuff in the interface.

Drawable d = new BouncingBox(...);
d.setMovementVectorVector(1, 1);

The method setMovementVector(int, int) is undefined for the type Drawable
Casting

If you know that a variable holds a specific type, you can use a cast:

Drawable d = new BouncingBox(…);
BouncingBox box = (BouncingBox) d;
box.setMovementVectorVector(1, 1);
Assignment: More graphics

Start a new project: code has changed.
6.092 Introduction to Programming in Java
January (IAP) 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.