

CSC 221: Computer Programming I

Fall 2009

Understanding class definitions

- class structure
- fields, constructors, methods
- parameters
- assignment statements
- local variables

Looking inside classes

recall that classes define the properties and behaviors of its objects

a class definition must:

- specify those properties and their types
- define how to create an object of the class
- define the behaviors of objects

FIELDS
CONSTRUCTOR
METHODS

```
public class CLASSNAME {  
    FIELDS  
  
    CONSTRUCTOR  
  
    METHODS  
}
```

mapping back to Scratch:

- *fields* are variables that belong to an individual sprite
- the *constructor* initializes the sprite's properties (like clicking the green flag)
- *methods* are scripts, executed by broadcasting a message

`public` is a visibility modifier – declaring the class to be public ensures that the user (and other classes) can use this class

Fields

fields store values for an object (a.k.a. instance variables)

- the collection of all fields for an object define its state
- when declaring a field, must specify its visibility, type, and name

```
private FIELD_TYPE FIELD_NAME;
```

for our purposes, all fields will be private (accessible to methods, but not to the user)

```
/**
 * A circle that can be manipulated and that draws itself on a canvas.
 *
 * @author Michael Kolling and David J. Barnes
 * @version 15 July 2000
 */

public class Circle {
    private int diameter;
    private int xPosition;
    private int yPosition;
    private String color;
    private boolean isVisible;

    . . .

}
```

text enclosed in `/** */` is a *comment* – visible to the user, but ignored by the compiler. Good for documenting code.

note that the fields are those values you see when you inspect an object in BlueJ

Constructor

a constructor is a special method that specifies how to create an object

- it has the same name as the class, public visibility (since called by the user)

```
public CLASS_NAME(OPTIONAL_PARAMETERS) {  
    STATEMENTS FOR INITIALIZING OBJECT STATE  
}
```

```
public class Circle {  
    private int diameter;  
    private int xPosition;  
    private int yPosition;  
    private String color;  
    private boolean isVisible;  
  
    /**  
     * Create a new circle at default position with default color.  
     */  
    public Circle() {  
        this.diameter = 30;  
        this.xPosition = 20;  
        this.yPosition = 60;  
        this.color = "blue";  
        this.isVisible = false;  
    }  
  
    . . .  
}
```

within a method, can refer to fields of the object via

`this.FIELD_NAME`

the period denotes ownership: you are referring to a field that belongs to "this" object

note: the 'this.' prefix is optional, but instructive

an *assignment statement* stores a value in a field

`this.FIELD_NAME = VALUE;`

here, default values are assigned for a circle

Methods

methods implement the behavior of objects

```
public RETURN_TYPE METHOD_NAME(OPTIONAL_PARAMETERS) {  
    STATEMENTS FOR IMPLEMENTING THE DESIRED BEHAVIOR  
}
```

```
public class Circle {  
    . . .  
  
    /**  
     * Make this circle visible. If it was already visible, do nothing.  
     */  
    public void makeVisible() {  
        this.isVisible = true;  
        this.draw();  
    }  
  
    /**  
     * Make this circle invisible. If it was already invisible, do nothing.  
     */  
    public void makeInvisible() {  
        this.erase();  
        this.isVisible = false;  
    }  
  
    . . .  
}
```

void return type specifies no value is returned by the method – here, the result is shown on the Canvas

note that one method can "call" another one

`this.draw()` calls the `draw` method on this circle

`this.erase()` calls the `erase` method on this circle

Simpler example: Die class

```
/**
 * This class models a simple die object, which can have any number of sides.
 *   @author Dave Reed
 *   @version 9/20/09
 */

public class Die {
    private int numSides;
    private int numRolls;

    /**
     * Constructs a 6-sided die object
     */
    public Die() {
        this.numSides = 6;
        this.numRolls = 0;
    }

    /**
     * Constructs an arbitrary die object.
     *   @param sides the number of sides on the die
     */
    public Die(int sides) {
        this.numSides = sides;
        this.numRolls = 0;
    }

    . . .
}
```

a `Die` object needs to keep track of its number of sides, number of times rolled

the default constructor (no parameters) creates a 6-sided die

can have multiple constructors (with parameters)

- a *parameter* is specified by its type and name
- a parameter represents a *temporary* value that can be used during the methods execution
- note: parameters are not prefixed with "this."

Simpler example: Die class (cont.)

```
. . .  
  
/**  
 * Gets the number of sides on the die object.  
 * @return the number of sides (an N-sided die can roll 1 through N)  
 */  
public int getNumberOfSides() {  
    return this.numSides;  
}  
  
/**  
 * Gets the number of rolls by on the die object.  
 * @return the number of times roll has been called  
 */  
public int getNumberOfRolls() {  
    return this.numRolls;  
}  
  
/**  
 * Simulates a random roll of the die.  
 * @return the value of the roll (for an N-sided die,  
 *         the roll is between 1 and N)  
 */  
public int roll() {  
    this.numRolls = this.numRolls + 1;  
    return (int)(Math.random()*this.numSides + 1);  
}  
}
```

a *return statement* specifies the value returned by a call to the method (shows up in a box in BlueJ)

a method that simply provides access to a private field is known as an *accessor method*

a method that changes the state is a *mutator* method

the `roll` method calculates a random roll (details later) and increments the number of rolls

PaperSheet example

```
public class PaperSheet {
    private double thickness;    // thickness in inches
    private int numFolds;       // the number of folds so far

    /**
     * Constructs the PaperSheet object
     * @param initial the initial thickness (in inches) of the paper
     */
    public PaperSheet(double initial) {
        this.thickness = initial;
        this.numFolds = 0;
    }

    /**
     * Folds the sheet, doubling its thickness as a result
     */
    public void fold() {
        this.thickness = 2 * this.thickness;
        this.numFolds = this.numFolds + 1;
    }

    /**
     * Repeatedly folds the sheet until the desired thickness is reached
     * @param goalDistance the desired thickness (in inches)
     */
    public void foldUntil(double goalDistance) {
        while (this.thickness < goalDistance) {
            this.fold();
        }
    }

    /**
     * Accessor method for determining folds
     * @return the number of times the paper has been folded
     */
    public int getNumFolds() {
        return this.numFolds;
    }
}
```

Another example: Singer

```
/**
 * This class can be used to display various children's songs.
 *   @author Dave Reed
 *   @version 9/20/09
 */
public class Singer
{
    /**
     * Constructor for objects of class Singer
     */
    public Singer() {
    }

    /**
     * Displays a verse of "OldMacDonald Had a Farm"
     *   @param animal animal name for this verse
     *   @param sound  sound that the animal makes
     */
    public void oldMacDonaldVerse(String animal, String sound) {
        System.out.println("Old MacDonald had a farm, E-I-E-I-O.");
        System.out.println("And on that farm he had a " + animal + ", E-I-E-I-O");
        System.out.println("With a " + sound + "-" + sound + " here, and a " +
            sound + "-" + sound + " there, ");
        System.out.println(" here a " + sound + ", there a " + sound +
            ", everywhere a " + sound + "-" + sound + ".");
        System.out.println("Old MacDonald had a farm, E-I-E-I-O.");
        System.out.println();
    }

    . . .
}
```

a `Singer` does not have any state, so no fields are needed

since no fields, constructor has nothing to initialize (should still have one, though)

`System.out.println` displays text in a window – can specify a String, a parameter name (in which case its value is displayed), or a combination using +

Another example: Singer (cont.)

```
. . .  
  
/**  
 * Displays the song "OldMacDonald Had a Farm"  
 */  
public void oldMacDonaldSong() {  
    this.oldMacDonaldVerse("cow", "moo");  
    this.oldMacDonaldVerse("duck", "quack");  
    this.oldMacDonaldVerse("sheep", "baa");  
    this.oldMacDonaldVerse("dog", "woof");  
}  
  
. . .  
  
}
```

one method can call another one:

```
this.METHOD_NAME(PARAMETERS)
```

again, the "this." prefix is optional, but instructive (emphasizes that the method is being called on *this* object)

when calling a method, the parameter values match up with the parameter names in the method based on order

```
this.oldMacDonaldVerse("cow", "moo");    → animal = "cow", sound = "moo"
```

```
this.oldMacDonaldVerse("meow", "cat");    → animal = "meow", sound =  
"cat"
```

HW3: experimentation with SequenceGenerator

add a method to the `SequenceGenerator` class to display multiple random sequences

```
/**
 * Displays a set number of random letter sequences of the specified length
 * @param numSequences the number of sequences to generate & display
 * @param seqLength the number of letters in the random sequences
 */
public void displaySequences(int numSequences, int seqLength) {
    int wordsPerLine = 40 / seqLength;

    int sequencesSoFar = 0;
    while (sequencesSoFar < numSequences) {
        System.out.print(this.randomSequence(seqLength) + " ");
        sequencesSoFar = sequencesSoFar + 1;
        if (sequencesSoFar % wordsPerLine == 0 || sequencesSoFar == numSequences) {
            System.out.println();
        }
    }
}
```

using this modified class, you will collect data to estimate the numbers of words with given characteristics

Examples from text: Bank Account & Cash Register

simple example: a bank account

- fields? account balance
- methods? construct an account (either with no money or a set amount)
 deposit a set amount
 withdraw a set amount

slightly more complex: a cash register

- fields? amount purchased (scanned) so far
 amount paid so far
- methods? construct a cash register
 purchase (scan) an item
 pay a set amount
 complete the purchase & get change

for now, we will assume the customer is honest

- customer will only enter positive amounts, will pay at least as much as purchase

CashRegister class

fields: maintain amounts
purchased and paid

constructor: initialize the
fields

methods: ???

```
/**
 * A cash register totals up sales and computes change due.
 * @author Dave Reed (based on code by Cay Horstmann)
 * @version 9/20/09
 */
public class CashRegister {
    private double purchase;
    private double payment;

    /**
     * Constructs a cash register with no money in it.
     */
    public CashRegister() {
        this.purchase = 0.0;
        this.payment = 0.0;
    }

    ...
}
```

CashRegister methods

recordPurchase:

- *mutator* method that returns adds to the purchase amount

enterPayment:

- *mutator* method that returns adds to the amount paid

giveChange:

- *mutator* method that returns the change owed to the customer (and resets the fields)

```
. . .

/**
 * Records the sale of an item.
 * @param amount the price of the item
 */
public void recordPurchase(double amount) {
    this.purchase = this.purchase + amount;
}

/**
 * Enters the payment received from the customer.
 * @param amount the amount of the payment
 */
public void enterPayment(double amount) {
    this.payment = this.payment + amount;
}

/**
 * Computes the change due and resets the machine
 * for the next customer.
 * @return the change due to the customer
 */
public double giveChange() {
    double change;
    change = this.payment - this.purchase;

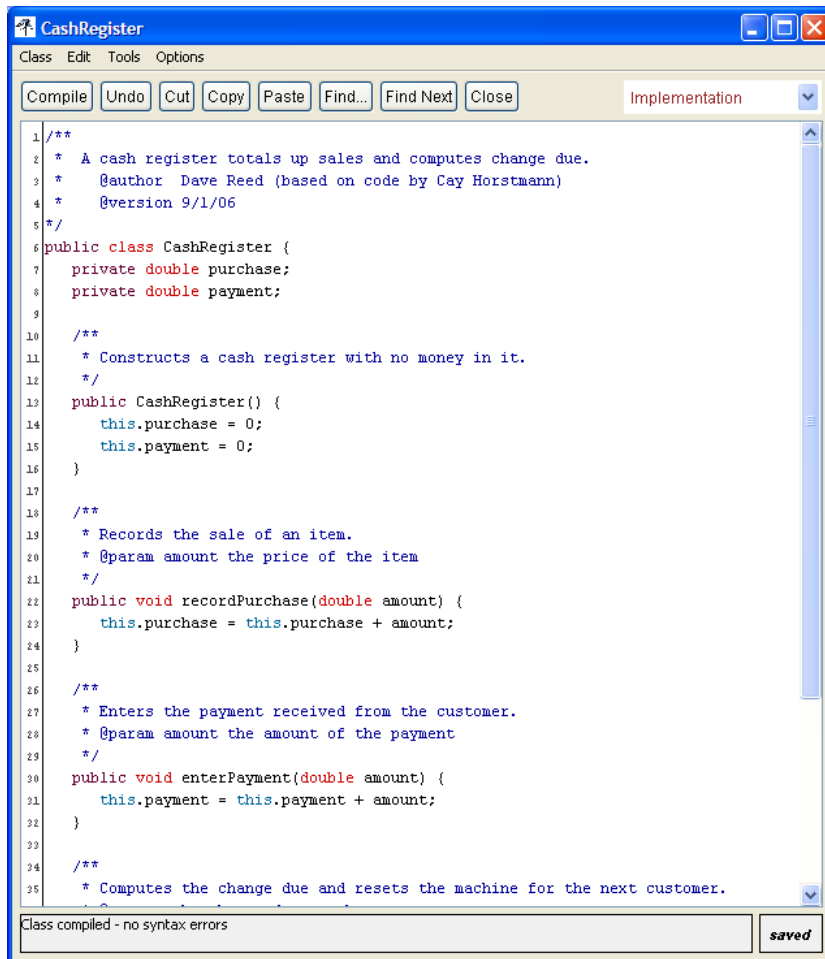
    this.purchase = 0;
    this.payment = 0;

    return change;
}
}
```

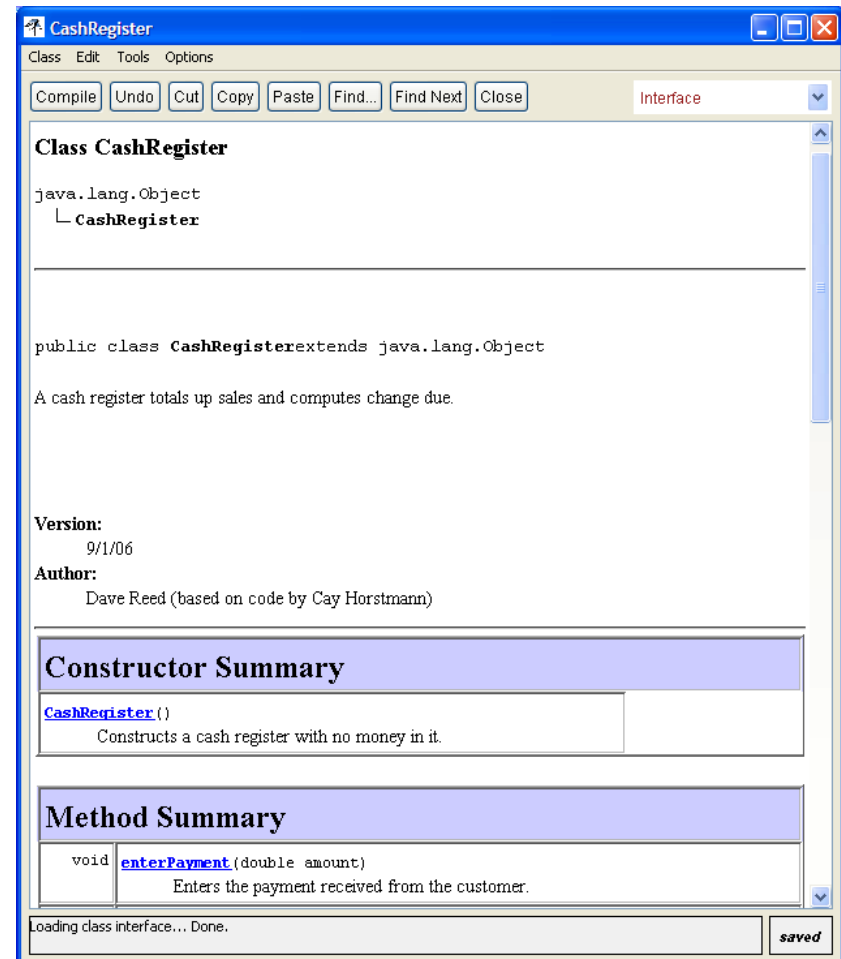
Interface view of class

comments that use `/** ... */` are *documentation comments*

- BlueJ will automatically generate a documentation page from these comments
- can view the documentation by selecting *Interface* from the top-right menu



```
1 /**
2  * A cash register totals up sales and computes change due.
3  * @author Dave Reed (based on code by Cay Horstmann)
4  * @version 9/1/06
5  */
6 public class CashRegister {
7     private double purchase;
8     private double payment;
9
10    /**
11     * Constructs a cash register with no money in it.
12     */
13    public CashRegister() {
14        this.purchase = 0;
15        this.payment = 0;
16    }
17
18    /**
19     * Records the sale of an item.
20     * @param amount the price of the item
21     */
22    public void recordPurchase(double amount) {
23        this.purchase = this.purchase + amount;
24    }
25
26    /**
27     * Enters the payment received from the customer.
28     * @param amount the amount of the payment
29     */
30    public void enterPayment(double amount) {
31        this.payment = this.payment + amount;
32    }
33
34    /**
35     * Computes the change due and resets the machine for the next customer.
```



Class CashRegister

java.lang.Object
└─ **CashRegister**

public class **CashRegister** extends java.lang.Object

A cash register totals up sales and computes change due.

Version:
9/1/06

Author:
Dave Reed (based on code by Cay Horstmann)

Constructor Summary

<code>CashRegister()</code>	Constructs a cash register with no money in it.
-----------------------------	---

Method Summary

void	<code>enterPayment(double amount)</code>	Enters the payment received from the customer.
------	--	--

Loading class interface... Done.

More on assignments

recall that fields are assigned values using an *assignment statement*

```
this.FIELD_NAME = VALUE;
```

field, parameter, method, class, and object names are all *identifiers*:

- can be any sequence of letters, underscores, and digits, but must start with a letter
e.g., `amount`, `recordPurchase`, `CashRegister`, `Circle`, `circle1`, ...

*by convention: class names start with capital letters; all others start with lowercase
when assigning a multiword name, capitalize inner words
avoid underscores (difficult to read in text)*

WARNING: capitalization matters, so `giveChange` and `givechange` are different!

each field in a class definition corresponds to a data value that must be stored for each object of that class

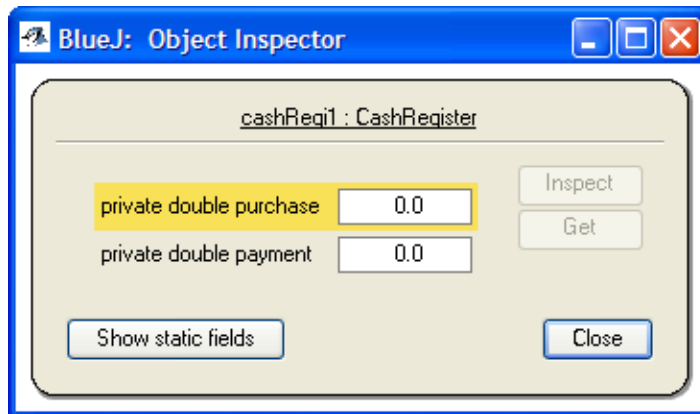
- when you create an object, memory is set aside to store that value
- when you perform an assignment, a value is stored in that memory location

Variables

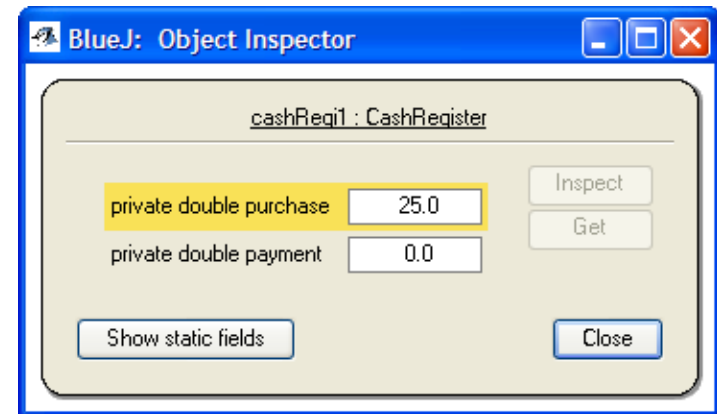
fields and parameters are examples of *variables*

- a *variable* is a name that refers to some value (which is stored in memory)
- when you assign a value to a variable, the Java interpreter finds its associated memory location and stores the value there
- if there was already a value there, it is overwritten

```
this.purchase = 0.0;
```



```
this.purchase = 25.0;
```



Assignments and expressions

the left-hand side of an assignment must be a variable (field or parameter);
the right-hand can be :

- a value (String, int, double, ...)

```
animal = "cow";           // the value on the right-hand side is  
this.payment = 0.0;      // assigned to and stored in the variable
```

- a variable (parameter or field name)

```
this.circleColor = color; // value represented by that variable is  
this.payment = amount;    // assigned to the field
```

- an expression using values, variables, and operators (+, -, *, /)

```
x = 2 + 3;                // can apply operators to values  
z = x - y;                // or variables  
this.num1 = this.num2 + 1; // or a combination
```

```
inchesToSun = 93000000.0 * 5280 * 12; // can use more than 1 operator
```

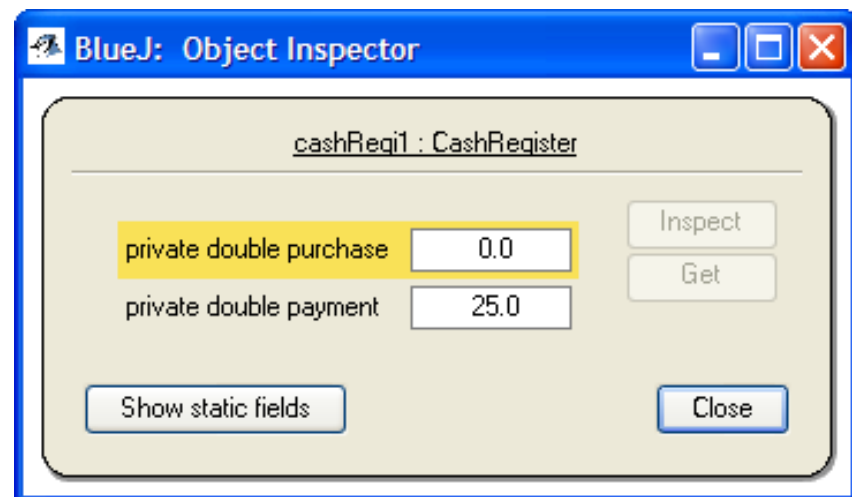
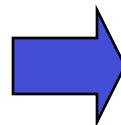
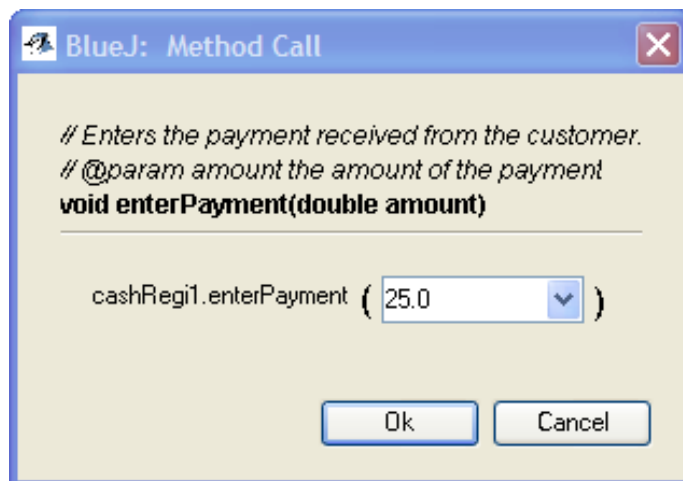
```
this.purchase = this.purchase + amount;  
// same variable can appear on both sides;  
// evaluate expression on right-hand side  
// using current value, then assign to left
```

More on parameters

recall that a parameter represents a value that is passed in to a method

- a parameter is a variable (it refers to a piece of memory where the value is stored)
- a parameter "belongs to" its method – only exists while that method executes
- using BlueJ, a parameter value can be entered by the user – that value is assigned to the parameter variable and subsequently used within the method

```
/**  
 * Enters the payment received from the customer.  
 * @param amount the amount of the payment  
 */  
public void enterPayment(double amount) {  
    this.payment = this.payment + amount;  
}
```



Local variables

fields are one sort of variable

- they store values through the life of an object
- they are accessible throughout the class

methods can include shorter-lived variables (called *local variables*)

- they exist only as long as the method is being executed
- they are only accessible from within the method
- local variables are useful whenever you need to store some temporary value (e.g., in a complex calculation)

before you can use a local variable, you must *declare it*

- specify the variable type and name (similar to fields, but no private modifier)

```
int num;  
String firstName;
```

- then, can use just like any other variable (assign it a value, print its value, ...)

Local variable example

```
public double giveChange() {  
    double change;  
    change = this.payment - this.purchase;  
  
    this.purchase = 0;  
    this.payment = 0;  
  
    return change;  
}
```

you can declare and assign a local variable at the same time

- preferable since it ensures you won't forget to initialize
- the compiler will complain if you try to access an uninitialized variable

```
public double giveChange() {  
    double change = this.payment - this.purchase;  
  
    this.purchase = 0;  
    this.payment = 0;  
  
    return change;  
}
```

When local variables?

local variables are useful when

- you need to store a temporary value (as part of a calculation or to avoid losing it)
- you are using some value over and over within a method

recall the `oldMacDonaldVerse` method from the `Singer` class

- what would have to change if we decided to spell the refrain "Eeyigh-eeyigh-oh" ?

```
public void oldMacDonaldVerse(String animal, String sound) {
    System.out.println("Old MacDonald had a farm, E-I-E-I-O.");
    System.out.println("And on that farm he had a " + animal + ", E-I-E-I-O.");
    System.out.println("With a " + sound + "-" + sound + " here, and a " +
        sound + "-" + sound + " there, ");
    System.out.println("  here a " + sound + ", there a " + sound +
        ", everywhere a " + sound + "-" + sound + ".");
    System.out.println("Old MacDonald had a farm, E-I-E-I-O.");
    System.out.println();
}
```

A better verse...

duplication within code is dangerous

- if you ever decide to change it, you must change it everywhere!

here, we could use a local variable to store the spelling of the refrain

- `System.out.println` will use this variable as opposed to the actual text
- if we decide to change the spelling, only one change is required (in the assignment)

```
public void oldMacDonaldVerse(String animal, String sound) {
    String refrain = "E-I-E-I-O"; // change the spelling here to affect entire verse

    System.out.println("Old MacDonald had a farm, " + refrain + ".");
    System.out.println("And on that farm he had a " + animal + ", " + refrain + ".");
    System.out.println("With a " + sound + "-" + sound + " here, and a " +
        sound + "-" + sound + " there, ");
    System.out.println(" here a " + sound + ", there a " + sound +
        ", everywhere a " + sound + "-" + sound + ".");
    System.out.println("Old MacDonald had a farm, " + refrain + ".");
    System.out.println();
}
```

Parameters vs. local variables

parameters are similar to local variables

- they only exist when that method is executing
- they are only accessible inside that method
- they are declared by specifying type and name (no private or public modifier)
- their values can be accessed/assigned within that method

however, they differ from local variables in that

- parameter declarations appear in the header for that method
- parameters are automatically assigned values when that method is called (based on the inputs provided in the call)

parameters and local variables both differ from fields in that they belong to (and are limited to) a method as opposed to the entire object

Quick-and-dirty summary

a class definition consists of: fields + constructors + methods

- *fields* are the data values that define the state of an object

```
private FIELD_TYPE FIELD_NAME;
```

- *constructors* initialize the state of an object when it is created
a class can have multiple constructors with different *parameters* to initialize the state differently
in its simplest form, a constructor contains *assignments* to fields

```
this.FIELD_NAME = VALUE_TO_BE_ASSIGNED;
```

- *methods* implement the behaviors or actions for an object
when defining a method, must specify its return type (or `void` if none) and parameters (if any)
 - a *return statement* is used to return a value computed by the method

```
return VALUE_TO_BE_RETURNED;
```

- a *print* or *println statement* is used to display text in a console window

```
System.out.print(TEXT_MESSAGE);      System.out.println(TEXT_MESSAGE);
```

fields & parameters are examples of *variables* (names that represent values)

- *fields* are variables that belong to an object, accessible by constructors & methods
- *parameters* are variables that are passed in and exist only inside a constructor/method
- in addition, *local variables* can be defined inside constructors/methods for temporary storage