CSCI 161
Introduction to Computer Science
Last Time...

We saw:

- How fields (instance variables) are declared
- How methods and constructors are defined
- How to create and call methods from BlueJ

Still not sure about:

- *When* to use local variables over fields?
- When to *return* a value from a method
- How to properly comment (document) your code
- How to make "branching" decisions?
- (Address these issues in this lecture)
Objects Can Simulate Real Things Right?

- Ticket machines can be found in subway and train stations

- What all TicketMachines do:
  - Have a set price/cost for a ticket
  - Prints a ticket when user inserts correct money
  - Keeps a running total of money collected
  - Assume: Only 1 ticket price and price is in whole dollars (too lazy to count cents)

- Let's define the TicketMachine class (in BlueJ!)
Demo

- **Readings:**
  - BlueJ book Chap 2

- Let's first take a look at a demonstration of how we expect the TicketMachine to behave

- Project is provided to you on course page
Outline

- Writing Our First Class: TicketMachine
  - Designing State
  - Designing Constructors
    - Parameters
  - Designing Methods
    - Printing to Screen
    - If-Statements
    - Getters and Setters
    - Local Variables
- Conclusion
Ticket Machine (State)

- Let's name the class `TicketMachine`

- What *state* should *all* ticket machines have?
  - What do they need to remember about themselves?
    - Amount of money inserted so far (we'll call that the *balance*)
    - Amount of money accumulated over time by the machine (called *total*)
    - Price or cost per ticket (called *price*)
  - *What are their data types?*
Outline

- Writing Our First Class: TicketMachine
  - Designing Fields
  - Designing Constructors
    - Parameters
  - Designing Methods
    - Printing to Screen
    - If-Statements
    - Getters and Setters
    - Local Variables
- Conclusion
 Constructors are responsible for setting up the ticket machine's state as soon as it's created

• Recall the syntax

```java
public ClassName(parameter_list) {
    //code to initialize object's state
}
```

• Remember, parameter lists might be empty

One Issue: Well... the balance and total should be zeroed out initially, but what about price per ticket?
Default Constructor

- **Default constructor**: a constructor that accepts no inputs
  - Also called the "no-arguments" (no-args) constructor

- **Know this**: Java automatically inserts a default constructor if you don't define it.
  - But it might not do what you want!
  - Better to be explicit: Write another constructor that takes no input parameters.
    - *In what initial state should it put newly-created TicketMachine objects?*
Outline

- Writing Our First Class: TicketMachine
  - Designing Fields
  - Designing Constructors
    - Parameters
  - Designing Methods
    - Printing to Screen
    - If-Statements
    - Getters and Setters
    - Local Variables

- Conclusion
What *behaviors* should our ticket machines have?

- Let a user `insertMoney`
- Retrieve the cost of a ticket: `getPrice`
- Retrieve the current balance: `getBalance`
- Let a user `printTicket`. It should also update total and clear the balance.

What can a ticket machine *do*?
Ticket Machine (Behaviors)

- For each method, ask:
  - Does the algorithm require any input? (Input params)
  - Does the algorithm owe the caller any values before it terminates? (Return type)
  - That's enough information for us to write the signature.
    - *Always write the method's signature first!!!*

- Recall the **template** for writing methods:

  ```java
  public returnType methodName(inputParamsIfAny) {
      statement 1;
      statement 2;
      ...
  }
  ```

  - **Signature**
  - **Body**
    - (contains the algorithm)
What Should \texttt{printTicket()} Do?

- **Step 1:** We want it to print the following to the screen:

  
  ```c
  #########################
  # The Puget Sound Line
  # Ticket
  # 5 dollars.
  #########################
  
  This number must reflect the cost of a single ticket at the particular machine
  \textit{(Hey, we have a field remembering that value)}

- **Step 2:** After printing, it should clear update the total and clear the balance.
Outline

- Writing Our First Class: TicketMachine
  - Designing Fields
  - Designing Constructors
    - Parameters
  - Designing Methods
    - Printing to Screen
    - If-Statements
    - Getters and Setters
    - Local Variables
- If-Statements
- Conclusion
How to Print Something to the Screen?

- **Syntax:**
  ```java
  System.out.println(stringWeWantPrinted);
  ```

  By the way, `stringWeWantPrinted` could also be a variable that's storing a String.

- **Examples:**
  ```java
  System.out.println("Hello World!");
  > Hello World      <------ This is what appears on the terminal!
  ```
  ```java
  String str = "Hello World!";
  System.out.println(str);
  > Hello World      <------ This is what appears on the terminal!
  ```
  ```java
  String str = "Hello\n\tWorld!";    // Note: "\n" means new line, "\t" means tab
  System.out.println(str);
  > Hello
  >     World
  ```
Important: Concatenating Strings!

- To *concatenate* is a fancy way of saying, "to append" Strings
  - We can append Strings to other Strings, math expressions, variables, etc.
  - The concatenation operator is: `+`

**Example:**
```java
int x = 100;
System.out.println("The value of x is "+ x);
> The value of x is 100
```

**Another:**
```java
int x = 13 * 4 + 9;
System.out.println("13 * 4 + 9 is\n:" + x);
> 13 * 4 + 9 is
> :61
```
Important: Concatenating Strings! (Cont.)

- Accumulator operator `+=` can also be used!

- Example: Build a String variable, then print it out!

```java
String str = "University\n";
str += "of\n";
str += "Puget Sound";
System.out.println(str);
```

```
University
of
Puget Sound
```
Finally... We Can Finish `printTicket()`

- The `printTicket()` method:
  - Step 1: We want it to print the following to the screen:

```
# The Puget Sound Line
# Ticket
# 5 dollars.
```

This must reflect the cost of a single ticket *(hey, we have a field for that!)*

- Step 2: After printing, it should update the total and the balance.
Outline

- Writing Our First Class: TicketMachine
  - Designing Fields
  - Designing Constructors
    - Parameters
  - Designing Methods
    - Printing to Screen
    - Getters and Setters
      - If-Statements
      - Local Variables

- Bugs

- Conclusion
Problem: Our TicketMachine has been losing tons of money and we don't know why!!!

Someone points out that our printTicket() method is broken... it allows you to print a ticket no matter how much $ you've put in.

Instead, we want it to make a choice

- Is the given balance enough to purchase a ticket?
  - If so, print a ticket
  - If not, print the amount that still needs entered
Re-Writing printTicket()

Currently... the printTicket() code looks like this:

```java
public void printTicket() {
    String ticket = "######################\n# The Puget Sound Line\n# Ticket\n# "+ price + " dollars.\n" + "######################"
    System.out.println(ticket);
    total += price;
    balance = 0;
}
```

What changes need to be made to this method:

- Is the balance enough to buy a ticket?
  - If so, print a ticket, accumulate the total, clear the balance just like before
  - If not, print the amount that still needs to be inserted
Conditional Statements

- What if we need to make a decision?
  - Enter: conditional statements

- This is known as an If-Then-Else:
  - The `else` block is entirely optional, but is needed in this case.

```java
if (true_or_false_condition) {
    //statements to execute if true
}
else {
    //statements to execute if false
}
```

(Okay, how do we re-write printTicket()?)
public void printTicket() {
    if (balance >= price) {
        String ticket = "#################################################################
";
        ticket += "# The Puget Sound Line\n";
        ticket += "# Ticket\n";
        ticket += "# " + price + " dollars.\n";
        ticket += "#################################################################
";
        System.out.println(ticket);

        balance -= price;
        total += price;
    } else { // they must not have entered enough
        System.out.println("You still owe " + (price - balance) + " dollars!");
    }
}
More Practice with If-Else

- We'll do this one together.

- What's the output when...
  - num1 = 2, num2 = 10
  - num1 = 10, num2 = 2
  - num1 = 2, num2 = 2

```java
if (num1 < num2) {
    System.out.println(" red ");
}
if ((num1 + 5) < num2) {
    System.out.println(" white ");
} else {
    System.out.println(" blue ");
}
System.out.println(" yellow ");
```
Your Turn (Code Reading)

- We'll do this one together.
- What's the output when...
  - num1 = 5, num2 = 4
  - num1 = 5, num2 = 12
  - num1 = 5, num2 = 27

```java
if (num1 >= num2) {
    System.out.println(" red ");
    System.out.print(" orange ");
}
if ((num1 + 5) >= num2) {
    System.out.println(" white ");
} else {
    if ((num1 + 10) >= num2) {
        System.out.println(" black ");
        System.out.println(" blue ");
    } else {
        System.out.println(" yellow ");
    }
}
System.out.println(" green ");
```
Your Turn (Code Writing)

- Write a method called `weather` that inputs an integer called `temperature` that will:
  - Print "It is cool" when `temperature` is less than or equal to 50.
  - Print "It is warm" on one line, and "dress coolly" on the next line, when `temperature` is greater than 80.
  - Print "It is pleasant" when `temperature` is between 50 and 80.

```java
public void weather(int temperature) {
    // One solution
    if (temperature <= 50) {
        System.out.println("It is cool");
    } else {
        if (temperature > 80) {
            System.out.println("It is warm. Dress coolly.");
        } else {
            System.out.println("It is pleasant");
        }
    }
}
```
Outline

- Writing Our First Class: TicketMachine
  - Designing Fields
  - Designing Constructors
    - Parameters
  - Designing Methods
    - Printing to Screen
    - If-Statements
    - Getters and Setters
    - Local Variables
- Scope and Lifetime of Variables
- Conclusion
Accessor Methods (Getters) are used to observe the object's current state

- They always return some value
- They have a return type that is non-void

- Really important (and worth repeating): returning is **not** the same as printing!

Let's take a closer look at `getPrice()` and `getBalance()`

```java
public int getPrice() {
    return price;
}
```

```java
public int getBalance() {
    return balance;
}
```
The Thing about return Statements

- **Important:** When a `return` statement is reached, any successive statements will not be executed!

(3) Control returns back here! `x` now has 80. Carry on with remaining code.

(1) Which method is being called? Execute it!

(2) Return statement reached, exit the method and return any value AND control to the caller!

```java
public int fizz() {
    int x;
    x = bizz(30, 50);
    //...
}

public int bizz(int x, int y) {
    System.out.println("foo");
    return (x + y);
    System.out.println("bar");
}  //dead code
```
Mutator Methods (Setters) are used to change an object's state

- They usually have a `void` return type
  - Just tweaking the object's current state;
  - Usually don't need to return any values

`insertMoney()` is an exception

- It's a setter that happens to return a value back to the caller

```java
public int insertMoney(int amount) {
    if (amount < 0) {
        System.out.println("Error: negative amount");
    } else {
        balance += amount;  //set the new balance here
    }
    return balance;
}
```
A New Method: Refund

- People have been requesting that our TicketMachine handle refunds!

- Think about what it needs to look like:
  - We can call the method `refundBalance()`
  - It returns the current balance
  - Then it clears the balance to zero

- What's wrong with the following code?

```java
public int refundBalance() {
    return balance; //return current balance to user
    balance = 0;    //clear the balance
}
```
Outline

- Writing Our First Class: TicketMachine
  - Designing Fields
  - Designing Constructors
    - Parameters
  - Designing Methods
    - Printing to Screen
    - If-Statements
    - Getters and Setters
      - Local Variables
- If-Statements
- Bugs
- Conclusion
Local Variables

- We need temporary storage to hold the `balance` before resetting the `balance` field.

```java
public int refundBalance()
{
    int amountToRefund;
    amountToRefund = balance;
    balance = 0; //clear the balance
    return amountToRefund; //return the amount to refund to user
}
```

A local variable (of type `int`) is declared here

- Methods and constructors can have their own "local variables"
  
  - Like input parameters, they only exist between the `{ . . . }`
  
  - Unlike input parameters, the body must set their values
  
  - They are only accessible within the method (or constructor)
You may have noticed BlueJ's highlighting scheme.

```java
/**
 * Print a ticket if enough money has been inserted, and
 * reduce the current balance by the ticket price. Print
 * an error message if more money is required.
 */
public void printTicket()
{
    if(balance >= price) {
        // Simulate the printing of a ticket.
        System.out.println("########################################");
        System.out.println("# The BlueJ Line");
        System.out.println("# Ticket");
        System.out.println("# " + price + " cents.");
        System.out.println("########################################");
        System.out.println();
        // Update the total collected with the price.
        total = total + price;
        // Reduce the balance by the price.
        balance = balance - price;
    } else {
        System.out.println("You must insert at least: " +
         (price - balance) + " more cents.");
    }
    /**
     * Return the money in the balance.
     * The balance is cleared.
     */
```
Remember those curly braces? They do more than just grouping statements together.

- Each "block of code" defines a new scope

**Important:** Scope and Lifetime of Variables

- The *scope of a variable* is the set of statements that can see and use it
- The *lifetime of a variable* is the boundaries of the variable's existence
Fields

- **Purpose**: Stores the state of an object
- **Who assigns their values**: Constructors and setters
- **Scope**: Visible & accessible by any method (or constructor) in the class
  - Any method can read from or store values into any field
- **Lifetime**: Persistent throughout the life of the object

```java
public class Person {
    //fields
    private boolean isFemale;
    private double height;
    private int age;
    private String eyeColor;

    public void setEyeColor(String color) {
        eyeColor = color;
    }
}
```
Input Parameters

- **Purpose:** Receive and store values from the method's caller
- **Who assigns their values:** Initially, the method's caller, but values can be reassigned by the method's statements
- **Scope:** Only visible to statements within the body of the method
- **Lifetime:** These variables are destroyed as soon as method finishes
  - Each call to the method receives a fresh set of values

```java
public class Person {
    //fields
    private boolean isFemale;
    private double height;
    private int age;
    private String eyeColor;

    public void setEyeColor(String color) {
        eyeColor = color;
    }
}
```
Local Variables

- **Purpose**: Methods often require temporary storage to complete its task
- **Who assigns their values**: Statements within the method body
  - Unlike parameters, not from outside
- **Scope**: Same as parameters
- **Lifetime**: Short-lived; same as parameters

```java
public class Rectangle {
    private double height;
    private double width;

    public double getArea() {
        double area = height * width;
        return area;
    }
}
```
What happens when you name a parameter the same name as a field?

- Java allows this! Poor design decision by Java, but I digress...

Important: When a variable name is ambiguous, Java uses the one closest in scope.
Outline

- Writing Our First Class: TicketMachine
  - Designing Fields
  - Designing Constructors
    - Parameters
  - Designing Methods
    - Printing to Screen
    - Getters and Setters
    - Local Variables
- If-Statements
- Conclusion
In Conclusion...

- **New concepts:**
  - Fields vs. parameters vs. local variables
    - Scope and lifetime
  - Methods: signature and body
    - Role of getters vs. setters
  - Printing to screen with: `System.out.println()`
  - If-then-Else Statement

- **Next time:** Defining New Classes & More on Conditionals (Chap 3 & 5)
Code for the updated Circle class is up on the course page
  • (The one we'd been working on last week)

Hwk 2 extended to next Wednesday

Lab 3 tomorrow: Stomach and Brain
  • Proper lab etiquette
    - Get here on time; read (not skim) the lab
    - Get off your phones and other unrelated web sites
Hwk 2 due tonight
  • Status? Questions?

Lab 4 tomorrow: All about if-then-else statements
  • Changes to Section CA Lab

Where were we before the snow came...?
  • TicketMachine class (take out your notes)
  • Writing Methods:
    - How to design methods
    - We talked about return types, and the return statement
    - We saw how to use if-then-else statements for insertMoney
Hwk 3 to be posted later today
  • TicketMachine lecture too!

Lab 4 post mortem:
  • Random number generation (and manipulation)
  • Math.abs() is useful!
  • Nesting if-then-else statements is a pain
    - Tracking lots of curly braces!!
    - Remember to auto-layout