Defining Classes

• Chap 1 introduced many concepts informally, in this chapter we will be more formal in defining
  – Classes, fields, and constructors
  – Methods and parameters, mutators and accessors
  – Assignment and conditional statements
• In addition to the topics noted above you will learn about
  – Blocks and the return statement
  – Compound assignment operators
  – Various forms of the if statement
  – Commenting your classes
The Example Projects

• A naïve ticket machine
  – It works correctly if all data is perfect
  – It does not guard against possible errors

• A better ticket machine
  – It guards against errors, such as specifying a negative value for the cost of a ticket
  – It introduces additional new concepts, such as local variables

• By the finish of this chapter you will be able to design and implement your own project
What problems did you notice with the naïve ticket machine?

• Run the naïve ticket machine program now
• Did you find any problems?

• We will fix these problems later, for now we will look at the source code for the naïve ticket machine
/**
 * TicketMachine models a naive ticket machine that issues flat-fare tickets.
 * The price of a ticket is specified via the constructor.
 * It is a naive machine in the sense that it trusts its users
 * to insert enough money before trying to print a ticket.
 * It also assumes that users enter sensible amounts.
 */

public class TicketMachine
{
    private int price; // The price of a ticket from this machine.
    private int balance; // The amount of money entered by a customer so far.
    private int total;   // The total amount of money collected by this machine.

    public TicketMachine(int ticketCost)
    {
        price = ticketCost;
        balance = 0;
        total = 0;
    }

    // methods go here
}

TicketMachine
Class
/**
 * Return the price of a ticket.
 */
public int getPrice()
{
    return price;
}

/**
 * Return the amount of money already inserted for the
 * next ticket.
 */
public int getBalance()
{
    return balance;
}

/**
 * Receive an amount of money in cents from a customer.
 */
public void insertMoney(int amount)
{
    balance += amount;
}
/**
 * Print a ticket.
 * Update the total collected and
 * reduce the balance to zero.
 */

public void printTicket()
{
    // 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 balance.
    total += balance;
    // Clear the balance.
    balance = 0;
}
The Structure of a Class Definition

```java
public class <name of class> {
    // declare fields of the class
    // declare constructor(s) for the class
    // declare methods for the class
}
```

- The word “public” indicates the visibility of the object, an alternative option is “private”
- Classes are usually declared to be public
- Fields are usually declared to be private; this means the values cannot be modified from outside of the class but only by methods inside the class
- Methods are public if they are called by other objects
- A method may be private if it is never used outside of the class; it is just a “helper” method inside the class
Declarations of Fields

- Field declarations have the form:
  ```java
  private <type> <name>;  as in
  private String major;
  ```
- Common types are int, double, char, boolean, and String (see next slide)
- The name typically starts with a lower case letter to distinguish from a class name, which starts uppercase
- Fields are almost always declared to be private
- It is possible to initialize a value at declaration, such as
  ```java
  private String major = “Computer Science”;
  ```
- A one line comment ( // …. ) can describe the use of a field in the context of the problem domain
Common Primitive Type

- "int" is an integer, that is, a signed whole number that has no decimal part
- "double" is a real number; it may be expressed in decimal notation or scientific notation
- "char" is a single character; Java uses Unicode to represent character values in 16 bits
- "boolean" only has two values: true and false
- "String" is not primitive, it is a Class that can hold a sequence of characters and has many methods for manipulating the string value
Constructors

- A constructor creates a new instance of a class and initializes values of the fields, as in
  
  ```java
  public TicketMachine(int ticketCost)
  {
      price = ticketCost;
      balance = 0;
      total = 0;
  }
  ```

- The name of the constructor **must** be the same as the class name; constructors do not return values.

- Constructors are typically public so that other classes can create instances of the class being defined.

- Each instance has a separate set of values for the fields of the class.

- A single class may have multiple constructors; give a constructor for TicketMachine that sets the price of a ticket to be $10, that is, 1000 cents.
Methods

- A typical method has the form

  `<visibility> <return type> <name> <parameters>`

  `{  
      <body of the method>
  }

  as in

  ```java
  public int getBalance()
  {
      return balance;
  }
  ```

  Notice there are no parameters

- The visibility is public or private
- The return type may be a primitive type, a class type (e.g. String), or void (nothing is returned)
- Method names start with lower case letters
- Parameters and body will be described shortly
Parameters

• A parameter list is a sequence of zero or more types and parameter names, as in:
  (int price)  // one item in this list
  (int price; int quantity)  // two items separated by ;
  ()  // this is an empty parameter list

• The type may be primitive (e.g., int, char) or a class type (e.g., String)

• The name is a formal parameter name that is used to access the data; the scope of visibility of the parameter is the body of the method

• The lifetime of the parameter ends when the method ends
Calling a Method

• When a method is called, values must be specified for each of the parameters

• The number of parameters and the sequence of types must match; when we use BlueJ, the values of parameters are entered in a popup window

• The objects or values passed in are referred to as the “actual” parameters

• If a value is returned by a method, it must be “received appropriately” by the calling program (more on this later)
The Assignment Statement - 1

• Assignment has the general form:

\(<\text{target variable}\> = \<\text{expression}\> \text{ such as }\)
\(\text{count} = \text{count} + 1;\)

• This assignment is performed as follows:
  – The expression on the right hand side (rhs) is evaluated
    • The current value of count is retrieved (it is an int)
    • The integer 1 is added to that value producing a new integer value
  – The result of the rhs expression is then stored in the variable count and becomes its new value

• Describe in simple English the result of performing this assignment
The Assignment Statement - 2

- The rhs expression has a type associated with it; in this example the type is int
- The type of the target variable should match the type of the rhs expression
- The target variable can be a field, a formal parameter, a local variable or any object visible in the current scope that is the appropriate type
- Note carefully that the use of “=” for assignment does not mean “equal” like in mathematics; shortly we will learn the equality relation is specified by “==”, two equal signs
Accessor Methods

• An accessor method fetches a value from the state of an instance of an object and returns it.

• For example

  ```java
  public int getBalance()
  {
    return balance;
  }
  ```

  is an accessor method that returns the current value stored in the field balance.

• Notice the following characteristics:
  – The method returns the type of the field being fetched.
  – Return is a statement; the expression to the right of the return is evaluated and the result is returned.
  – It is a convention to name accessor methods with the prefix “get” before the name of the field.
Mutator Methods - 1

• A mutator is designed to change the state of an instance of an object; usually only a single field is changed, for example

```java
public void insertMoney(int amount) {
    balance += amount;
}
```

is a mutator that changes the value stored in the field balance by adding amount to the balance.

• Notice the following characteristics:
  – The method returns void
  – The assignment is a compound assignment that is shorthand for writing `balance = balance + amount;`
Mutator Methods - 2

- Often times a mutator assigns a parameter value directly to a field, such as

  ```java
  public void setBalance(int amount) {
    balance = amount;
  }
  ```

- This is informally called a “setter” method; we will often talk about “getter/setter” methods for a field that is part of an object.

- Notice the following about lifetimes:
  - The lifetime of the formal parameter `amount` is the lifetime of the body of the method `setBalance`.
  - The lifetime of the field `balance` is the lifetime of the instance of the class `TicketMachine`.
Printing Output

- The printTicket method displays textual information about an instance of TicketMachine
- `System.out.println( "<value to be printed">) outputs a String value to the text terminal in BlueJ, as in
  ```java
  System.out.println("# Ticket");
  ```
  Printing the literal string “# Ticket” then going to the next line (the –ln suffix can be omitted if desired)
- The actual string expression must be evaluated before sending the string to the text terminal
  ```java
  System.out.println("# " + price + " cents.");
  ```
  - “+” is the String concatenation operator
  - When an int value, such as price, is concatenated, the internal representation of int value is converted to a String representation of that value that is then concatenated
  - If price had the value 500, then “# 500 cents” would be printed
Review of TicketMachine

- We now go back to the source code for the TicketMachine and discuss the code formally
- At the same time, keep track of things that might “go wrong” using this code
- << review the code now >>
- What things did you find as potential problems?
Potential Problems

- The program does not check if the customer underpays
- The program does not give a refund if the customer overpays
- It does not check if the amounts of money are “sensible” (what do you think is sensible?)
- It does not check if the ticket price is sensible

To fix these problems we will introduce
- A conditional statement (the “if” we got a preview of in chapter 1)
- Declaring local variables in a method body
A Better Ticket Machine

• Run the project in BlueJ first and see how it performs

• A conditional statement has the form:

```java
if (<<evaluate a boolean expr>>) {
    <<do what is here if boolean expr is true>>
} else {
    <<do what is here if boolean expr is false>>
}
```
Boolean Expressions

• A Boolean expression returns a value of true or false; unlike C++, values such as 0 and 1 are not valid Boolean expressions.

• For numbers, a Boolean expression is often a comparison based on the relations `==` (equal), `<`, `<=`, `>`, `>=` and `!=` (not equal).

• The String class has several Boolean methods, assume the String variable name has a value
  – `name.equals("Barry")` returns true or false
  – `name.startsWith("A")` returns true or false
  – `name.endsWith("field")` returns true or false
Revised insertMoney Method

```java
public void insertMoney(int amount) {
    if(amount > 0) {
        balance += amount;
    }
    else {
        System.out.println("Use a positive amount: "+ amount);
    }
}
```

- If the amount is positive, this works the same as before
- If the amount is zero or negative, the balance isn’t changed and a warning message is printed
Revised printTicket Method

```java
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 += price;
        // Reduce the balance by the price.
        balance -= price;
    }
    else {
        System.out.println("You must insert at least: " +
                          (price - balance) + " more cents.");
    }
}
```

- This version makes sure the customer paid enough, otherwise a warning message is printed
refundBalance Method

/**
 * Return the money in the balance.
 * The balance is cleared.
 */

public int refundBalance()
{
    int amountToRefund;
    amountToRefund = balance;
    balance = 0;
    return amountToRefund;
}

• Notice the declaration of the local variable amountToRefund

• Why is this variable even needed in the program?
Fields, Parameters, Local Variables

• Lifetimes
  – Fields exist as long as the instance of the object exists
  – Parameters and local variables exist only as long as the method that contains them is executing its code

• Scope
  – The scope of fields is the entire class
  – The scope of local variables and parameters is the method in which they are declared

• Parameters differ from local variables in an important way: parameters are initialized to the actual parameter values while local variables need to be explicitly initialized