CSC 222: Object-Oriented Programming

Fall 2015

Understanding class definitions

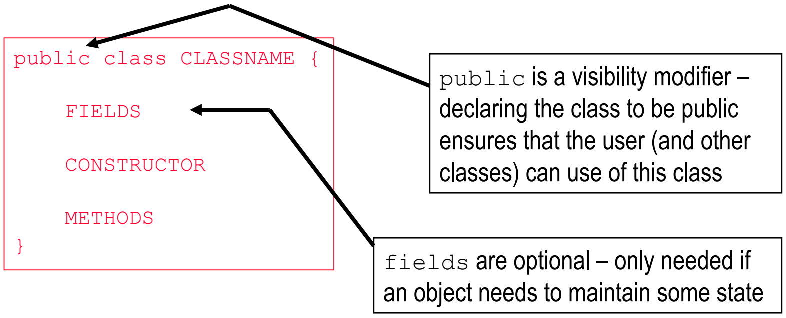
- class structure
- fields, constructors, methods
- parameters
- assignment statements
- conditional statements
- local variables

Looking inside classes

recall that classes define the properties and behaviors of its objects

a class definition must:

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



```
public class CLASSNAME {

    FIELDS

    CONSTRUCTOR

    METHODS
}
```

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

`fields` are optional – only needed if an object needs to maintain some state

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 2011.07.31
 */

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() {
    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;
    }

    . . .
}
```

an *assignment statement* stores a value in a field

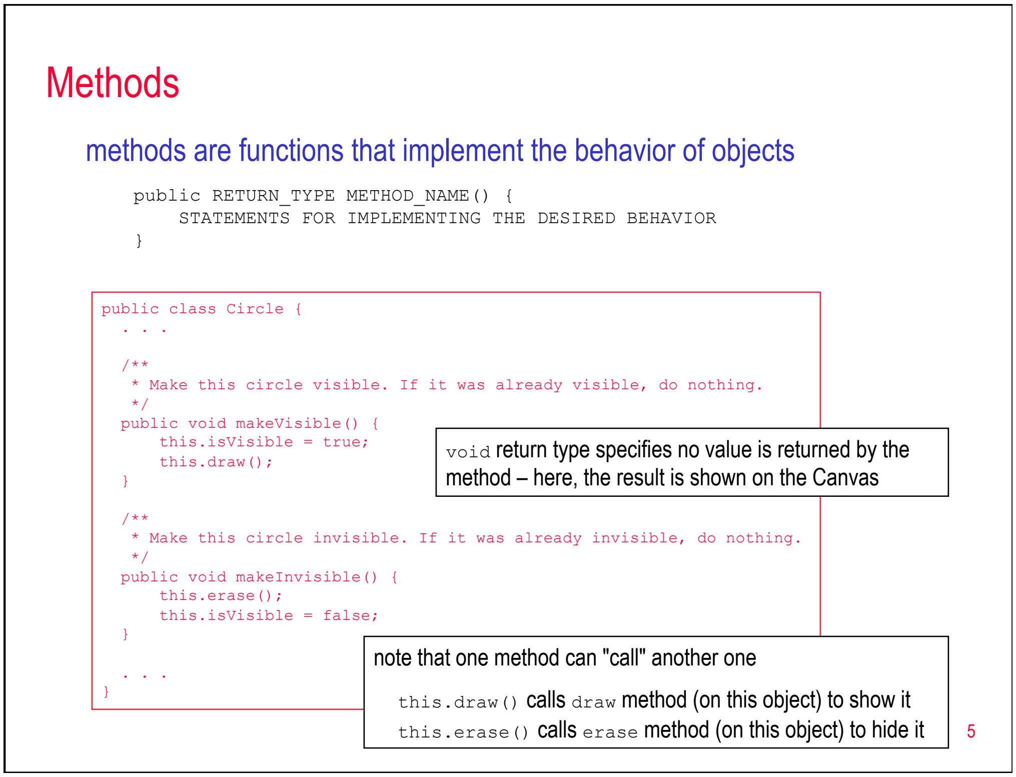
```
this.FIELD_NAME = VALUE;
```

here, default values are assigned for a circle

when referring to a field, the `this.` prefix is optional

- makes it clear that the variable is a field & belongs to this particular object
- I will use consistently in my code & strongly recommend you do so too

NOTE: all Java statements (incl. field declarations and assignments) end with a semi-colon



Methods

methods are functions that implement the behavior of objects

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

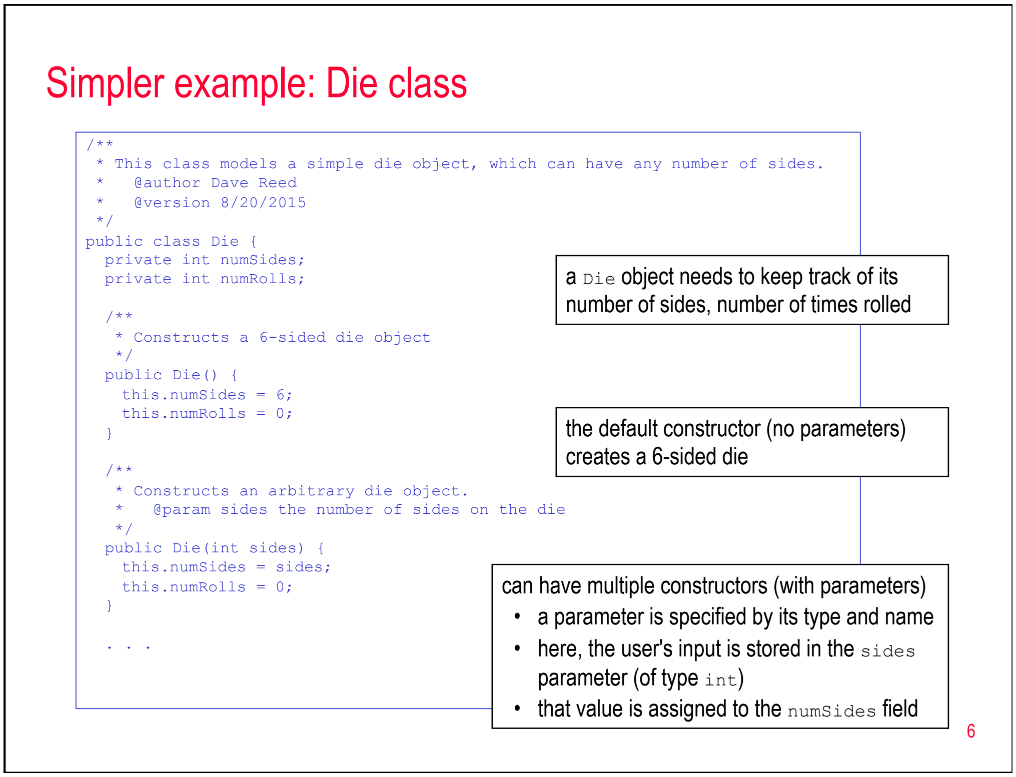
```
public class Circle {
  . . .

  /**
   * 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 draw method (on this object) to show it

this.erase() calls erase method (on this object) to hide it



Simpler example: Die class

```
/**
 * This class models a simple die object, which can have any number of sides.
 *   @author Dave Reed
 *   @version 8/20/2015
 */
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
- here, the user's input is stored in the sides parameter (of type int)
- that value is assigned to the numSides field

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*

the `roll` method calculates a random rolls and increments the number of rolls

Java class vs. Python class

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

  public Die(int sides) {
    this.numSides = sides;
    this.numRolls = 0;
  }

  public int roll() {
    this.numRolls = this.numRolls + 1;
    return (int)(Math.random()*this.numSides) + 1;
  }

  public int getNumSides() {
    return this.numSides;
  }

  public int getNumRolls() {
    return this.numRolls;
  }
}
```

```
class Die:
    def __init__(self, sides):
        self.numSides = sides
        self.numRolls = 0

    def roll(self):
        self.numRolls = self.numRolls+1
        return randint(1, self.numSides)

    def getNumSides(self):
        return self.numSides

    def getNumRolls(self):
        return self.numRolls
```

- Java provides accessibility options (all public in Python), so must specify
- Java variables (fields & parameters) are committed to one type of value, so must declare name & type
- methods that return values must specify the type
- constructor is similar to `__init__`
- `this.` in Java is similar to `self.` (but don't specify as method parameter)

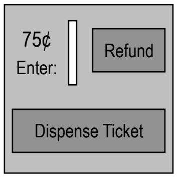
Alternate constructor

```
/**
 * This class models a simple die object, which can have any number of sides.
 *   @author Dave Reed
 *   @version 8/20/2015
 */

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

  /**
   * Constructs a 6-sided die object
   */
  public Die() {
    this(6);
  }

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

  . . .
```

just as a method can call another method, a constructor can call another constructor

we could have written the default constructor so that it called the other constructor, specifying 6 as the # of sides

Example from text: Ticket Machine



consider a simple ticket machine

- machine supplies tickets at a fixed price
- customer enters cash, possibly more than one coin in succession
- when correct amount entered, customer hits a button to dispense ticket
  →if not enough entered, no ticket + instructions to add more money
- customer can receive a refund if overpaid by hitting another button
- machine keeps track of total sales for the day

for now, we will assume the customer is honest

- customer will only enter positive amounts
- customer only hits the button when the exact amount has been inserted
- so, don't need to worry about checking the amount or giving change (for now)

TicketMachine class

fields: will need to store
- price of the ticket
- amount entered so far by the customer
- total intake for the day

constructor: will take the fixed price of a ticket as parameter
- must initialize the fields

insertMoney:
- *mutator* method that adds to the customer's balance

```
/**
 * This class simulates a simple ticket vending machine.
 *   @author  Barnes & Kolling (minor edits by Reed)
 *   @version 7/31/2015
 */

public class TicketMachine {
  private int price;          // price of a ticket
  private int balance;        // amount entered by user
  private int total;          // total intake for the day

  /**
   * Constructs a ticket machine.
   *   @param ticketCost the fixed price of a ticket
   */
  public TicketMachine(int ticketCost) {
    this.price = ticketCost;
    this.balance = 0;
    this.total = 0;
  }

 /**
   * Mutator method for inserting money to the machine.
   *   @param amount the amount of cash (in cents)
   *          inserted by the customer
   */
  public void insertMoney(int amount)
  {
    this.balance = this.balance + amount;
  }
  . . .
}
```

Note: // is used for inline comments (everything following // on a line is ignored by the compiler)

TicketMachine methods

getPrice:
- *accessor* method that returns the ticket price

getBalance:
- *accessor* method that returns the balance

printTicket:
- simulates the printing of a ticket (assuming correct amount has been entered)

```
/**
  * Accessor method for the ticket price.
  *   @return the fixed price (in cents) for a ticket
  */
 public int getPrice() {
   return this.price;
 }

/**
  * Accessor method for the amount inserted.
  *   @return the amount inserted so far
  */
 public int getBalance() {
   return this.balance;
 }

 /**
  * Simulates the printing of a ticket.
  *   Note: this naive method assumes that the user
  *   has entered the correct amount.
  */
 public void printTicket() {
   System.out.println("##################");
   System.out.println("# The BlueJ Line");
   System.out.println("# Ticket");
   System.out.println("# " + this.price + " cents.");
   System.out.println("##################");
   System.out.println();

   // Update the total collected & clear the balance.
   this.total = this.total + this.balance;
   this.balance = 0;
 }
}
```

Variables & assignments

fields and parameters are examples of *variables*

- a *variable* is a name that refers to some value (which is stored in memory)
- variables are assigned values using an *assignment statement*

```
VARIABLE = VALUE;
```

- variable names can be any sequence of letters, underscores, and digits, but must start with a letter

  e.g., `amount`, `balance`, `getPrice`, `TicketMachine`, …

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 `getPrice` *and* `getprice` *are different names!*

Assignments and expressions

the right-hand side of an assignment can be

▪ a value (`String`, `int`, `double`, …)	`this.balance = 0;`
▪ a variable (parameter or field name)	`this.price = cost;`
▪ an expression using (+, -, *, /)	`this.balance = this.balance + amount;`

updating an existing value is a fairly common task, so *arithmetic assignments* exist as shorthand notations:

`x += y;`	is equivalent to	`x = x + y;`		
`x -= y;`	is equivalent to	`x = x - y;`		
`x *= y;`	is equivalent to	`x = x * y;`		
`x /= y;`	is equivalent to	`x = x / y;`		
`x++;`	is equivalent to	`x = x + 1;`	is equivalent to	`x += 1;`
`x--;`	is equivalent to	`x = x - 1;`	is equivalent to	`x -= 1;`

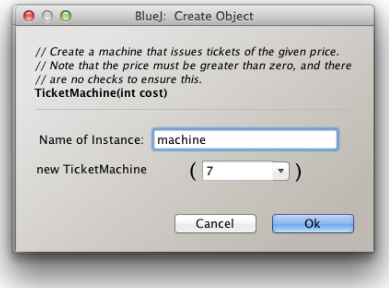
when + is applied to a String, values are concatenated end-to-end

```
System.out.println("Hi " + "there.");    // outputs "Hi there"

System.out.println("sum = " + (1 + 2));  // outputs "sum = 3"
```

More on parameters

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

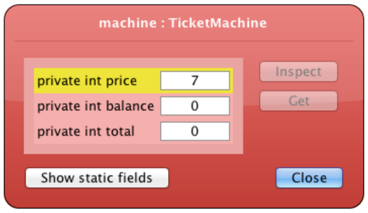
- a parameter is a variable (it refers to a piece of memory where the value is stored)
- a parameter "belongs to" its method – only exists while that method executes
- using BlueJ, a parameter value can be entered by the user – that value is assigned to the parameter variable and subsequently used within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



```
// Create a machine that issues tickets of the given price.
// Note that the price must be greater than zero, and there
// are no checks to ensure this.
TicketMachine(int cost)
```

Name of Instance: machine

new TicketMachine (7)



private int price	7
private int balance	0
private int total	0

Extending the Ticket Machine

now let us consider a fully-functional ticket machine

- will need to make sure that the user can't insert a negative amount
- will need to make sure that the amount entered >= ticket price before dispensing
- will need to give change if customer entered too much

all of these involve *conditional* actions

- add amount to balance *only if it is positive*
- dispense ticket *only if amount entered >= ticket price*
- give change *only if amount entered > ticket price*

in Java, an *if statement* allows for conditional execution

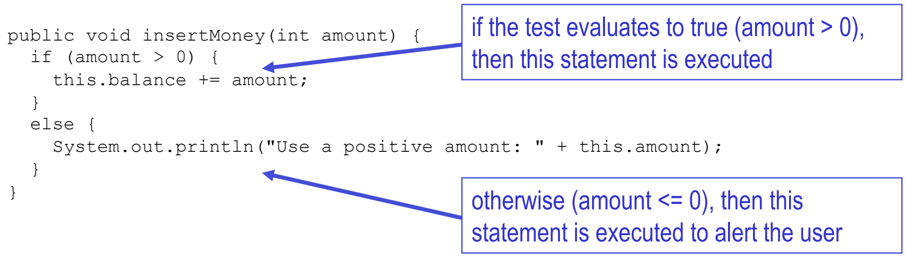
- i.e., can choose between 2 alternatives to execute

```
if (perform some test) {
    Do the statements here if the test is true
}
else {
    Do the statements here if the test is false
}
```

unlike Python, indentation is not meaningful to the interpreter (braces determine structure)

use indentation to make intent clear!

Conditional execution via if statements



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

if the test evaluates to true (amount > 0), then this statement is executed

otherwise (amount <= 0), then this statement is executed to alert the user

you are not required to have an else case to an if statement

- if no else case exists and the test evaluates to false, nothing is done
- e.g., could have just done the following

```
public void insertMoney(int amount) {
  if(amount > 0) {
    this.balance += amount;
  }
}
```

but then no warning to user if a negative amount were entered (not as nice)

17

Relational operators

standard relational operators are provided for the if test

<	less than	>	greater than
<=	less than or equal to	>=	greater than or equal to
==	equal to	!=	not equal to

a comparison using a relational operator is known as a *Boolean expression*, since it evaluates to a *Boolean* (`true` or `false`) value

```
public void printTicket() {
  if (this.balance >= this.price) {
    System.out.println("##################");
    System.out.println("# The BlueJ Line");
    System.out.println("# Ticket");
    System.out.println("# " + this.price + " cents.");
    System.out.println("##################");
    System.out.println();

    // Update the total collected & clear the balance.
    this.total += this.price;
    this.balance -= this.price;
  }
  else {
    System.out.println("You must enter at least: " +
                       (this.price - this.balance) + " cents.");
  }
}
```

18

Multiway conditionals

a simple if statement is a 1-way conditional (*execute this or not*)

```
if (number < 0) {                  // if number is negative
    number = -1 * number;          //   will make it positive
}                                  // otherwise, do nothing and move on
```

an if statement with else case is a 2-way conditional (*execute this or that*)

```
if (number < 0) {                            // if number is negative
    System.out.println("negative");          //   will display "negative"
}                                            //
else {                                       // otherwise,
    System.out.println("non-negative");      //   will display "non-negative"
}
```

if more than 2 possibilities, can "cascade" if statements

```
if (number < 0) {
  System.out.println("negative");
}
else {
  if (number > 0) {
    System.out.println("positive");
  }
  else {
    System.out.println("zero");
  }
}
```

can omit some braces and line up neatly:

```
if (number < 0) {
  System.out.println("negative")
}
else if (number > 0) {
    System.out.println("positive");
}
else {
    System.out.println("zero");
}
```

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

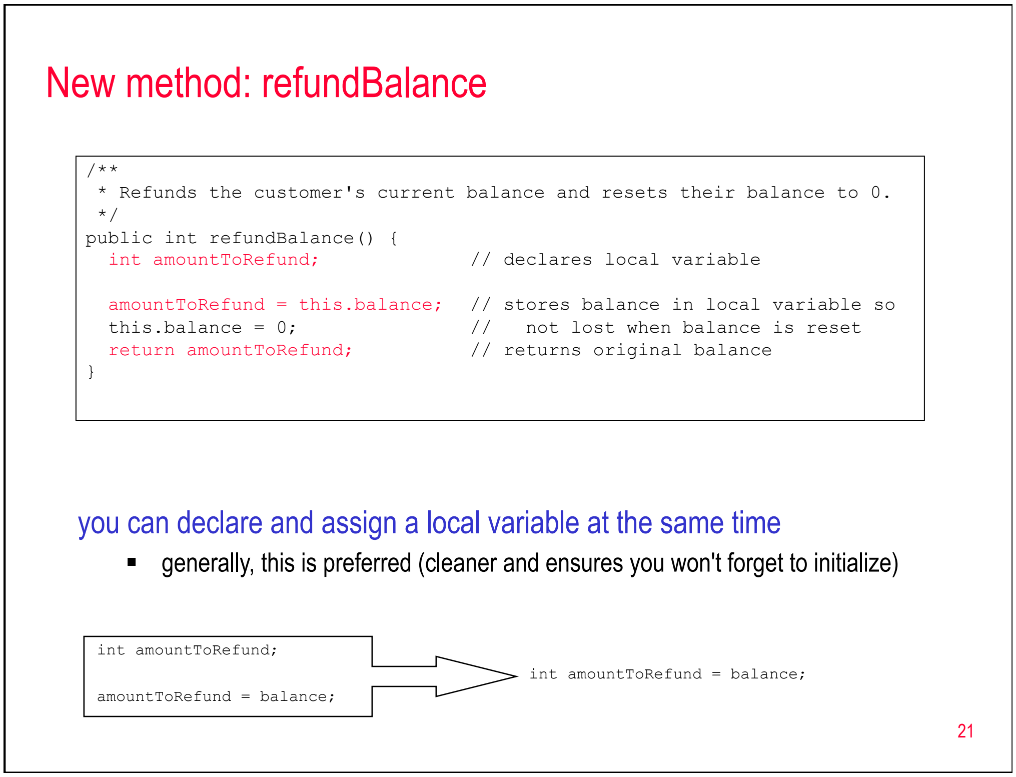
- local variables are useful whenever you need to store some temporary value (e.g., in a complex calculation)

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, …)



New method: refundBalance

```
/**
 * Refunds the customer's current balance and resets their balance to 0.
 */
public int refundBalance() {
  int amountToRefund;             // declares local variable

  amountToRefund = this.balance;  // stores balance in local variable so
  this.balance = 0;               //   not lost when balance is reset
  return amountToRefund;          // returns original balance
}
```

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

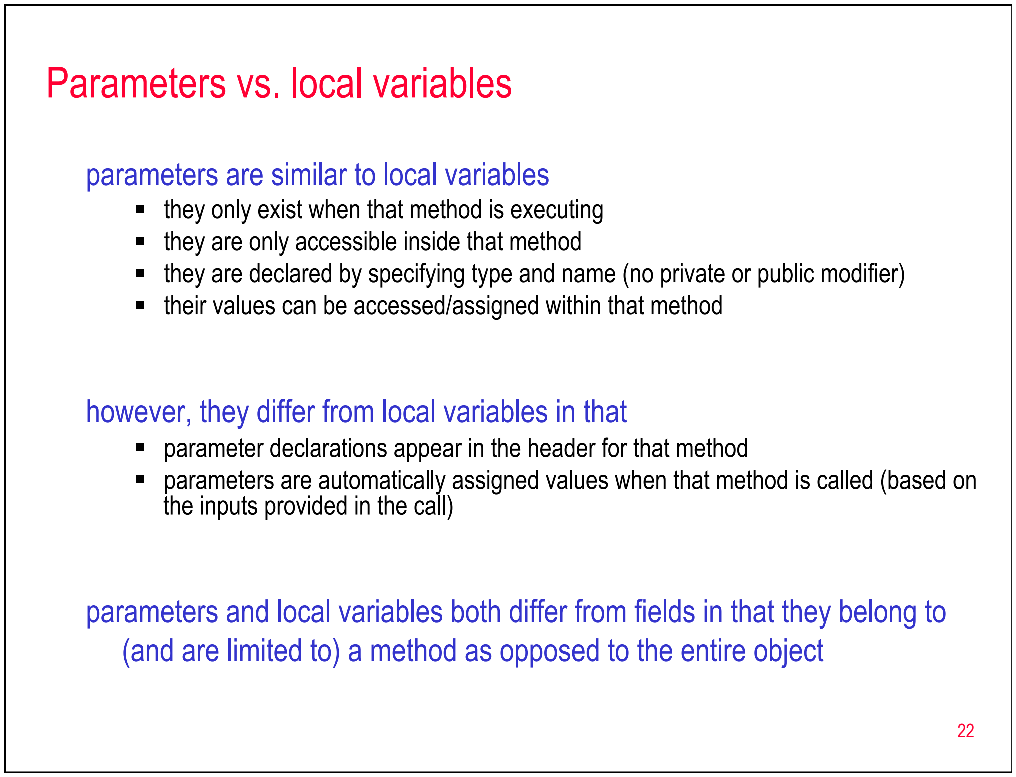
- generally, this is preferred (cleaner and ensures you won't forget to initialize)

```
int amountToRefund;

amountToRefund = balance;
```

→

```
int amountToRefund = balance;
```

21



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

22