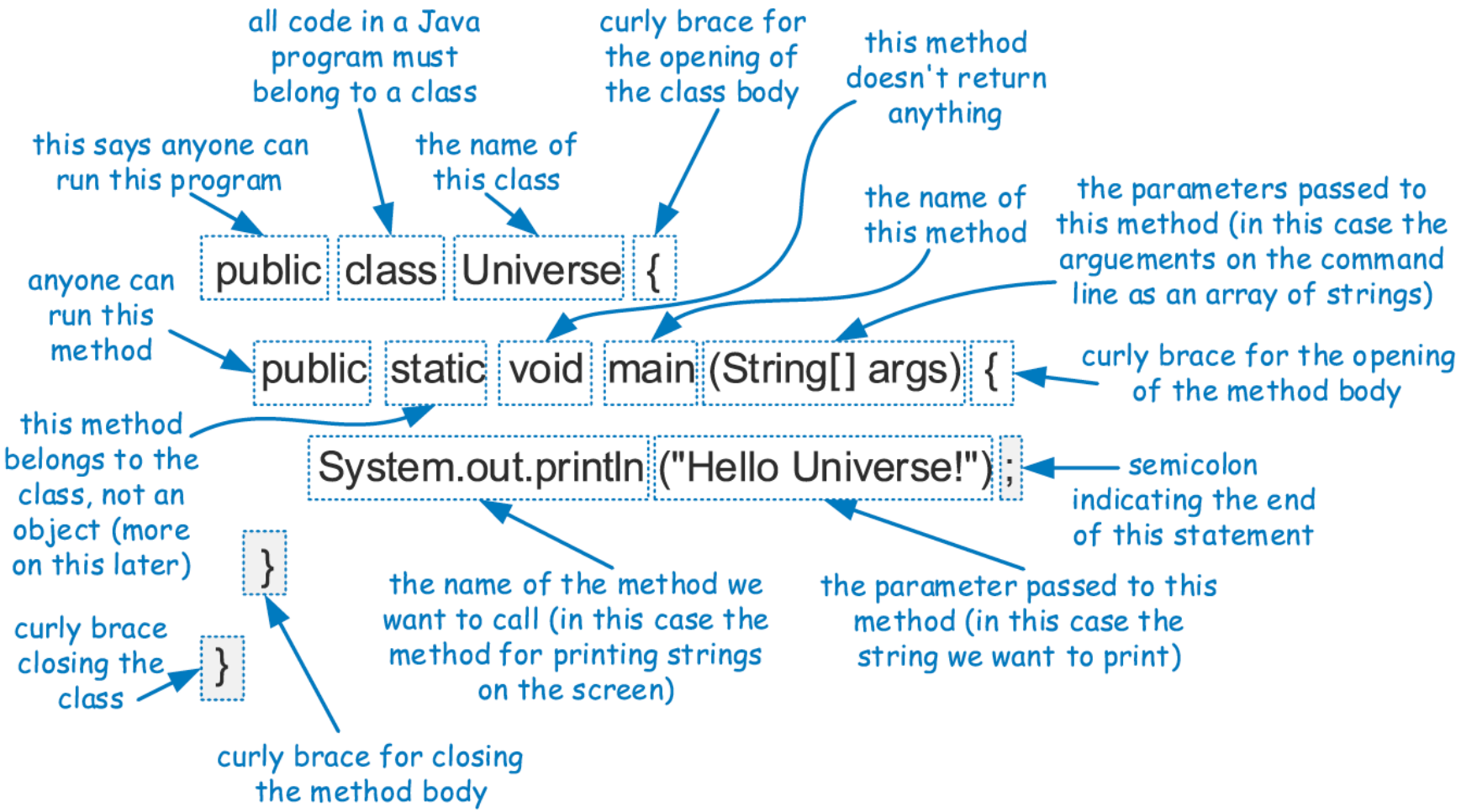
JAVA PRIMER:
TYPES, CLASSES AND OPERATORS
I/O METHODS AND CONTROL FLOW

Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

THE JAVA COMPILER

- Java is a compiled language.
- Programs are compiled into byte-code executable files, which are executed through the Java virtual machine (JVM).
  - The JVM reads each instruction and executes that instruction.
- A programmer defines a Java program in advance and saves that program in a text file known as source code.
- For Java, source code is conventionally stored in a file named with the **.java** suffix (e.g., **demo.java**) and the byte-code file is stored in a file named with a **.class** suffix, which is produced by the Java compiler.

AN EXAMPLE PROGRAM



COMPONENTS OF A JAVA PROGRAM

- In Java, executable statements are placed in functions, known as **methods**, that belong to class definitions.
- The static method named **main** is the first method to be executed when running a Java program.
- Any set of statements between the braces "**{**" and "**}**" define a program block.

IDENTIFIERS

- The name of a class, method, or variable in Java is called an **identifier**, which can be any string of characters as long as it begins with a letter and consists of letters.
- Exceptions:

Reserved Words				
abstract	default	goto	package	synchronized
assert	do	if	private	this
boolean	double	implements	protected	throw
break	else	import	public	throws
byte	enum	instanceof	return	transient
case	extends	int	short	true
catch	false	interface	static	try
char	final	long	strictfp	void
class	finally	native	super	volatile
const	float	new	switch	while
continue	for	null		

BASE TYPES

- Java has several base types, which are basic ways of storing data.
- An identifier variable can be declared to hold any base type and it can later be reassigned to hold another value of the same type.

boolean	a boolean value: true or false
char	16-bit Unicode character
byte	8-bit signed two's complement integer
short	16-bit signed two's complement integer
int	32-bit signed two's complement integer
long	64-bit signed two's complement integer
float	32-bit floating-point number (IEEE 754-1985)
double	64-bit floating-point number (IEEE 754-1985)

```
boolean flag = true;
boolean verbose, debug;
char grade = 'A';
byte b = 12;
short s = 24;
int i, j, k = 257;
long l = 890L;
float pi = 3.1416F;
double e = 2.71828, a = 6.022e23;
```

CLASSES AND OBJECTS

- Every **object** is an instance of a **class**, which serves as the type of the object and as a blueprint, defining the data which the object stores and the methods for accessing and modifying that data. The critical members of a class in Java are the following:
  - **Instance variables**, which are also called **fields**, represent the data associated with an object of a class. Instance variables must have a type, which can either be a base type (such as int, float, or double) or any class type.
  - **Methods** in Java are blocks of code that can be called to perform actions. Methods can accept parameters as arguments, and their behavior may depend on the object upon which they are invoked and the values of any parameters that are passed. A method that returns information to the caller without changing any instance variables is known as an **accessor** method, while an **update** method is one that may change one or more instance variables when called.

ANOTHER EXAMPLE

```
public class Counter {
  private int count;                                  // a simple integer instance variable
  public Counter( ) { }                               // default constructor (count is 0)
  public Counter(int initial) { count = initial; }    // an alternate constructor
  public int getCount( ) { return count; }            // an accessor method
  public void increment( ) { count++; }               // an update method
  public void increment(int delta) { count += delta; }// an update method
  public void reset( ) { count = 0; }                 // an update method
}
```

- This class includes one instance variable, named count, which will have a default value of zero, unless we otherwise initialize it.
- The class includes two special methods known as constructors, one accessor method, and three update methods.

CREATING AND USING OBJECTS

- Classes are known as **reference types** in Java, and a variable of that type is known as a **reference variable**.
- A reference variable is capable of storing the location (i.e., memory address) of an object from the declared class.
  - So we might assign it to reference an existing instance or a newly constructed instance.
  - A reference variable can also store a special value, null, that represents the lack of an object.
- In Java, a new object is created by using the **new** operator followed by a call to a constructor for the desired class.
- A **constructor** is a method that always shares the same name as its class. The new operator returns a reference to the newly created instance; the returned reference is typically assigned to a variable for further use.

CONTINUED EXAMPLE

```
public class CounterDemo {
  public static void main(String[ ] args) {
    Counter c;                          // declares a variable; no counter yet constructed
    c = new Counter( );                 // constructs a counter; assigns its reference to c
    c.increment( );                     // increases its value by one
    c.increment(3);                     // increases its value by three more
    int temp = c.getCount( );           // will be 4
    c.reset( );                         // value becomes 0
    Counter d = new Counter(5);         // declares and constructs a counter having value 5
    d.increment( );                     // value becomes 6
    Counter e = d;                      // assigns e to reference the same object as d
    temp = e.getCount( );               // will be 6 (as e and d reference the same counter)
    e.increment(2);                     // value of e (also known as d) becomes 8
  }
}
```

- Here, a new Counter is constructed at line 4, with its reference assigned to the variable c. That relies on a form of the constructor, Counter( ), that takes no arguments between the parentheses.

THE DOT OPERATOR

- One of the primary uses of an object reference variable is to access the members of the class for this object, an instance of its class.
- This access is performed with **the dot (“.”) operator.**
- We call a method associated with an object by using the reference variable name, following that by the dot operator and then the method name and its parameters.

```
e.increment(2);
```

WRAPPER TYPES

- There are many data structures and algorithms in Java's libraries that are specifically designed so that they only work with object types (not primitives).
- To get around this obstacle, Java defines a **wrapper** class for each base type.
  - Java provides additional support for implicitly converting between base types and their wrapper types through a process known as automatic **boxing** and **unboxing**.

EXAMPLE WRAPPER TYPES

Base Type	*Class Name*	*Creation Example*	*Access Example*
boolean	Boolean	obj = new Boolean(true);	obj.booleanValue()
char	Character	obj = new Character('Z');	obj.charValue()
byte	Byte	obj = new Byte((byte) 34);	obj.byteValue()
short	Short	obj = new Short((short) 100);	obj.shortValue()
int	Integer	obj = new Integer(1045);	obj.intValue()
long	Long	obj = new Long(10849L);	obj.longValue()
float	Float	obj = new Float(3.934F);	obj.floatValue()
double	Double	obj = new Double(3.934);	obj.doubleValue()

```
int j = 8;
Integer a = new Integer(12);
int k = a;                          // implicit call to a.intValue()
int m = j + a;                      // a is automatically unboxed before the addition
a = 3 * m;                          // result is automatically boxed before assignment
Integer b = new Integer("-135");    // constructor accepts a String
int n = Integer.parseInt("2013");   // using static method of Integer class
```

SIGNATURES

- If there are several methods with this same name defined for a class, then the Java runtime system uses the one that matches the actual number of parameters sent as arguments, as well as their respective types.

- A method's name combined with the number and types of its parameters is called a method's **signature**, for it takes all of these parts to determine the actual method to perform for a certain method call.

- A reference variable **v** can be viewed as a **"pointer"** to some object **o**.

DEFINING CLASSES

- A **class definition** is a block of code, delimited by braces “{” and “}” , within which is included declarations of instance variables and methods that are the members of the class.

- Immediately before the definition of a class, instance variable, or method in Java, keywords known as **modifiers** can be placed to convey additional stipulations about that definition.

ACCESS CONTROL MODIFIERS

- The **public** class modifier designates that all classes may access the defined aspect.
- The **protected** class modifier designates that access to the defined aspect is only granted to classes that are designated as subclasses of the given class through inheritance or in the same package.
- The **private** class modifier designates that access to a defined member of a class be granted only to code within that class.
- When a variable or method of a class is declared as **static**, it is associated with the class as a whole, rather than with each individual instance of that class.

PARAMETERS

- A method's parameters are defined in a comma-separated list enclosed in parentheses after the name of the method.
  - A parameter consists of two parts, the parameter type and the parameter name.
  - If a method has no parameters, then only an empty pair of parentheses is used.
- All parameters in Java are **passed by value**, that is, any time we pass a parameter to a method, a copy of that parameter is made for use within the method body.
  - So if we pass an int variable to a method, then that variable's integer value is copied.
  - The method can change the copy but not the original.
  - If we pass an object reference as a parameter to a method, then the reference is copied as well.

THE KEYWORD THIS

- Within the body of a method in Java, the keyword **this** is automatically defined as a reference to the instance upon which the method was invoked. There are three common uses:
  1. To store the reference in a variable, or send it as a parameter to another method that expects an instance of that type as an argument.
  2. To differentiate between an instance variable and a local variable with the same name.
  3. To allow one constructor body to invoke another constructor body.

ARITHMETIC OPERATORS

- Java supports the following arithmetic operators:

+	addition
−	subtraction
*	multiplication
/	division
%	the modulo operator

- If both operands have type int, then the result is an int; if one or both operands have type float, the result is a float.
- Integer division has its result truncated.

INCREMENT AND DECREMENT OPERATORS

- Java provides the plus-one increment (++) and decrement (−−) operators.
  - If such an operator is used in front of a variable reference, then 1 is added to (or subtracted from) the variable and its value is read into the expression.
  - If it is used after a variable reference, then the value is first read and then the variable is incremented or decremented by 1.

```
int i = 8;
int j = i++;          // j becomes 8 and then i becomes 9
int k = ++i;          // i becomes 10 and then k becomes 10
int m = i--;          // m becomes 10 and then i becomes 9
int n = 9 + --i;      // i becomes 8 and then n becomes 17
```

LOGICAL OPERATORS

- Java supports the following operators for numerical valu es, which result in Boolean values:

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

- Boolean values also have the following operators:

!	not (prefix)
&&	conditional and
\|\| | conditional or

- The and and or operators **short circuit**, in that they do not evaluate the second operand if the result can be det ermined based on the value of the first operand.

BITWISE OPERATORS

- Java provides the following bitwise operators for integers and booleans:

~	bitwise complement (prefix unary operator)
&	bitwise and
|	bitwise or
^	bitwise exclusive-or
<<	shift bits left, filling in with zeros
>>	shift bits right, filling in with sign bit
>>>	shift bits right, filling in with zeros

OPERATOR PRECEDENCE

Operator Precedence		
	Type	**Symbols**
1	array index method call dot operator	[ ] ( ) .
2	postfix ops prefix ops cast	*exp*++ *exp*−− ++*exp* −−*exp* +*exp* −*exp* ~*exp* !*exp* (*type*) *exp*
3	mult./div.	* / %
4	add./subt.	+ −
5	shift	<< >> >>>
6	comparison	< <= > >= **instanceof**
7	equality	== !=
8	bitwise-and	&
9	bitwise-xor	^
10	bitwise-or	|
11	and	&&
12	or	||
13	conditional	*booleanExpression* ? *valueIfTrue* : *valueIfFalse*
14	assignment	= += −= *= /= %= <<= >>= >>>= &= ^= |=

CASTING

- Casting is an operation that allows us to change the type of a value.
- We can take a value of one type and cast it into an equivalent value of another type.
- There are two forms of casting in Java: **explicit casting** and **implicit casting.**

EXPLICIT CASTING

- Java supports an explicit casting syntax with the following form:

  (type) exp

- Here "type" is the type that we would like the expression exp to have.
- This syntax may only be used to cast from one primitive type to another primitive type, or from one reference type to another reference type.
- Examples:

```
double d1 = 3.2;
double d2 = 3.9999;
int i1 = (int) d1;                  // i1 gets value 3
int i2 = (int) d2;                  // i2 gets value 3
double d3 = (double) i2;            // d3 gets value 3.0
```

IMPLICIT CASTING

- There are cases where Java will perform an implicit cast based upon the context of an expression.
- You can perform a **widening cast** between primitive types (such as from an int to a double), without explicit use of the casting operator.
- However, if attempting to do an implicit **narrowing cast**, a compiler error results.

```
int i1 = 42;
double d1 = i1;                    // d1 gets value 42.0
i1 = d1;                           // compile error: possible loss of precision
```

IF STATEMENTS

- The syntax of a simple **if** statement is as follows:

```
if (booleanExpression)
    trueBody
else
    falseBody
```

- booleanExpression is a boolean expression and trueBody and falseBody are each either a single statement or a block of statements enclosed in braces (“{” and “}”).

COMPOUND IF STATEMENTS

- There is also a way to group a number of boolean tests, as follows:

```
if (firstBooleanExpression)
    firstBody
else if (secondBooleanExpression)
    secondBody
else
    thirdBody
```

SWITCH STATEMENTS

- Java provides for multiple-value control flow using the switch statement.
- The switch statement evaluates an integer, string, or enum expression and causes control flow to jump to the code location labeled with the value of this expression.
- If there is no matching label, then control flow jumps to the location labeled “default.”
- This is the only explicit jump performed by the switch statement, however, so flow of control “falls through” to the next case if the code for a case is not ended with a **break** statement

SWITCH EXAMPLE

```
switch (d) {
  case MON:
    System.out.println("This is tough.");
    break;
  case TUE:
    System.out.println("This is getting better.");
    break;
  case WED:
    System.out.println("Half way there.");
    break;
  case THU:
    System.out.println("I can see the light.");
    break;
  case FRI:
    System.out.println("Now we are talking.");
    break;
  default:
    System.out.println("Day off!");
}
```

BREAK AND CONTINUE

- Java supports a **break** statement that immediat ely terminate a while or for loop when executed within its body.
- Java also supports a **continue** statement that ca uses the current iteration of a loop body to stop, but with subsequent passes of the loop proceed ing as expected.

WHILE LOOPS

- The simplest kind of loop in Java is a **while** loop.
- Such a loop tests that a certain condition is satisfied and will perform the body of the loop each time this condition is evaluated to be true.
- The syntax for such a conditional test before a loop body is executed is as follows:

while (booleanExpression)

loopBody

DO-WHILE LOOPS

- Java has another form of the while loop that allows the boolean condition to be checked at the end of each pass of the loop rather than before each pass.
- This form is known as a do-while loop, and has syntax shown below:

```
do
        loopBody
while (booleanExpression)
```

FOR LOOPS

- The traditional **for**-loop syntax consists of four sections—an initialization, a boolean condition, an increment statement, and the body—although any of those can be empty.
- The structure is as follows:

  for (initialization; booleanCondition; increment)

  loopBody

- Meaning:

```
{
    initialization;
    while (booleanCondition) {
        loopBody;
        increment;
    }
}
```

EXAMPLE FOR LOOPS

- Compute the sum of an array of doubles:

```
public static double sum(double[ ] data) {
  double total = 0;
  for (int j=0; j < data.length; j++)      // note the use of length
    total += data[j];
  return total;
}
```

- Compute the maximum in an array of doubles:

```
public static double max(double[ ] data) {
  double currentMax = data[0];        // assume first is biggest (for now)
  for (int j=1; j < data.length; j++)      // consider all other entries
    if (data[j] > currentMax)              // if data[j] is biggest thus far...
      currentMax = data[j];                // record it as the current max
  return currentMax;
}
```

FOR-EACH LOOPS

- Since looping through elements of a collection is such a common construct, Java provides a shorthand notation for such loops, called the **for-each** loop.
- The syntax for such a loop is as follows:

```
for (elementType name : container)
    loopBody
```

FOR-EACH LOOP EXAMPLE

- Computing a sum of an array of doubles:

```
public static double sum(double[ ] data) {
  double total = 0;
  for (double val : data)                    // Java's for-each loop style
    total += val;
  return total;
}
```

- When using a for-each loop, there is no explicit use of array indices.
- The loop variable represents one particular element of the array.

SIMPLE OUTPUT

- Java provides a built-in static object, called System.out, t hat performs output to the “standard output” device, wit h the following methods:

print(String *s*):	Print the string *s*.
print(Object *o*):	Print the object *o* using its toString method.
print(*baseType b*):	Print the base type value *b*.
println(String *s*):	Print the string *s*, followed by the newline character.
println(Object *o*):	Similar to print(*o*), followed by the newline character.
println(*baseType b*):	Similar to print(*b*), followed by the newline character.

JAVA.UTIL.SCANNER METHODS

- The Scanner class reads the input stream and divides it into tokens, which are strings of characters separated by delimiters.

hasNext( ):	Return **true** if there is another token in the input stream.
next( ):	Return the next token string in the input stream; generate an error if there are no more tokens left.
hasNext*Type*( ):	Return **true** if there is another token in the input stream and it can be interpreted as the corresponding base type, *Type*, where *Type* can be Boolean, Byte, Double, Float, Int, Long, or Short.
next*Type*( ):	Return the next token in the input stream, returned as the base type corresponding to *Type*; generate an error if there are no more tokens left or if the next token cannot be interpreted as a base type corresponding to *Type*.

SIMPLE INPUT

- There is also a special object, **System.in**, for performing input from t he Java console window.
- A simple way of reading input with this object is to use it to create a **Scanner** object, using the expression

  new Scanner(System.in)

- Example:

```
import java.util.Scanner;                    // loads Scanner definition for our use

public class InputExample {
  public static void main(String[ ] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter your age in years: ");
    double age = input.nextDouble( );
    System.out.print("Enter your maximum heart rate: ");
    double rate = input.nextDouble( );
    double fb = (rate − age) ∗ 0.65;
    System.out.println("Your ideal fat-burning heart rate is " + fb);
  }
}
```

SAMPLE PROGRAM

```
1  public class CreditCard {
2    // Instance variables:
3    private String customer;        // name of the customer (e.g., "John Bowman")
4    private String bank;            // name of the bank (e.g., "California Savings")
5    private String account;         // account identifier (e.g., "5391 0375 9387 5309")
6    private int limit;              // credit limit (measured in dollars)
7    protected double balance;       // current balance (measured in dollars)
8    // Constructors:
9    public CreditCard(String cust, String bk, String acnt, int lim, double initialBal) {
10     customer = cust;
11     bank = bk;
12     account = acnt;
13     limit = lim;
14     balance = initialBal;
15   }
16   public CreditCard(String cust, String bk, String acnt, int lim) {
17     this(cust, bk, acnt, lim, 0.0);                  // use a balance of zero as default
18   }
```

```
19    // Accessor methods:
20    public String getCustomer( ) { return customer; }
21    public String getBank( ) { return bank; }
22    public String getAccount( ) { return account; }
23    public int getLimit( ) { return limit; }
24    public double getBalance( ) { return balance; }
25    // Update methods:
26    public boolean charge(double price) {          // make a charge
27      if (price + balance > limit)                 // if charge would surpass limit
28        return false;                              // refuse the charge
29      // at this point, the charge is successful
30      balance += price;                            // update the balance
31      return true;                                 // announce the good news
32    }
33    public void makePayment(double amount) {       // make a payment
34      balance -= amount;
35    }
36    // Utility method to print a card's information
37    public static void printSummary(CreditCard card) {
38      System.out.println("Customer = " + card.customer);
39      System.out.println("Bank = " + card.bank);
40      System.out.println("Account = " + card.account);
41      System.out.println("Balance = " + card.balance);   // implicit cast
42      System.out.println("Limit = " + card.limit);       // implicit cast
43    }
44    // main method shown on next page...
45  }
```

```
1   public static void main(String[ ] args) {
2     CreditCard[ ] wallet = new CreditCard[3];
3     wallet[0] = new CreditCard("John Bowman", "California Savings",
4                                "5391 0375 9387 5309", 5000);
5     wallet[1] = new CreditCard("John Bowman", "California Federal",
6                                "3485 0399 3395 1954", 3500);
7     wallet[2] = new CreditCard("John Bowman", "California Finance",
8                                "5391 0375 9387 5309", 2500, 300);
9
10    for (int val = 1; val <= 16; val++) {
11      wallet[0].charge(3*val);
12      wallet[1].charge(2*val);
13      wallet[2].charge(val);
14    }
15
16    for (CreditCard card : wallet) {
17      CreditCard.printSummary(card);          // calling static method
18      while (card.getBalance( ) > 200.0) {
19        card.makePayment(200);
20        System.out.println("New balance = " + card.getBalance( ));
21      }
22    }
23  }
```