EECS 2101

Fundamentals of

Data Structures

Instructor: Andy Mirzaian

Course Topics

- Fundamental **data structures** underlying widely-used algorithms:
  - arrays, lists,
    maps, hash tables,
    priority queues,
    search trees,
    graphs
  - and their algorithmic applications

- Covered in an **Object-Oriented** context

- Using the **Java** Programming Language

Course Outcomes

By the end of the course, students will be familiar with the more prevalent data structure patterns, and will be able to design and implement variations on these patterns, and use them to solve a broad range of real-world problems.

Administrivia

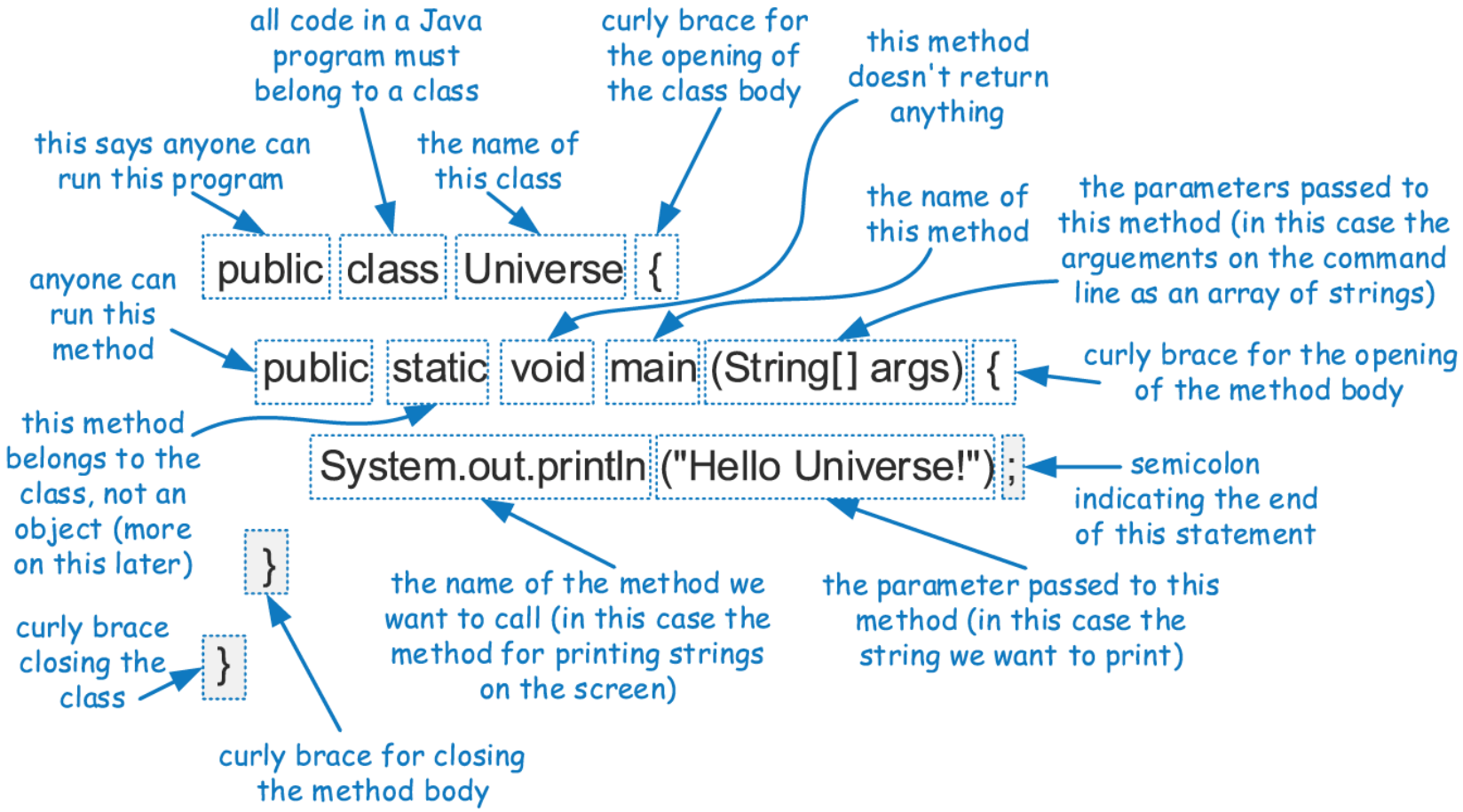
Go to eClass

Java Primer: Part 1
Types, Classes & Operators

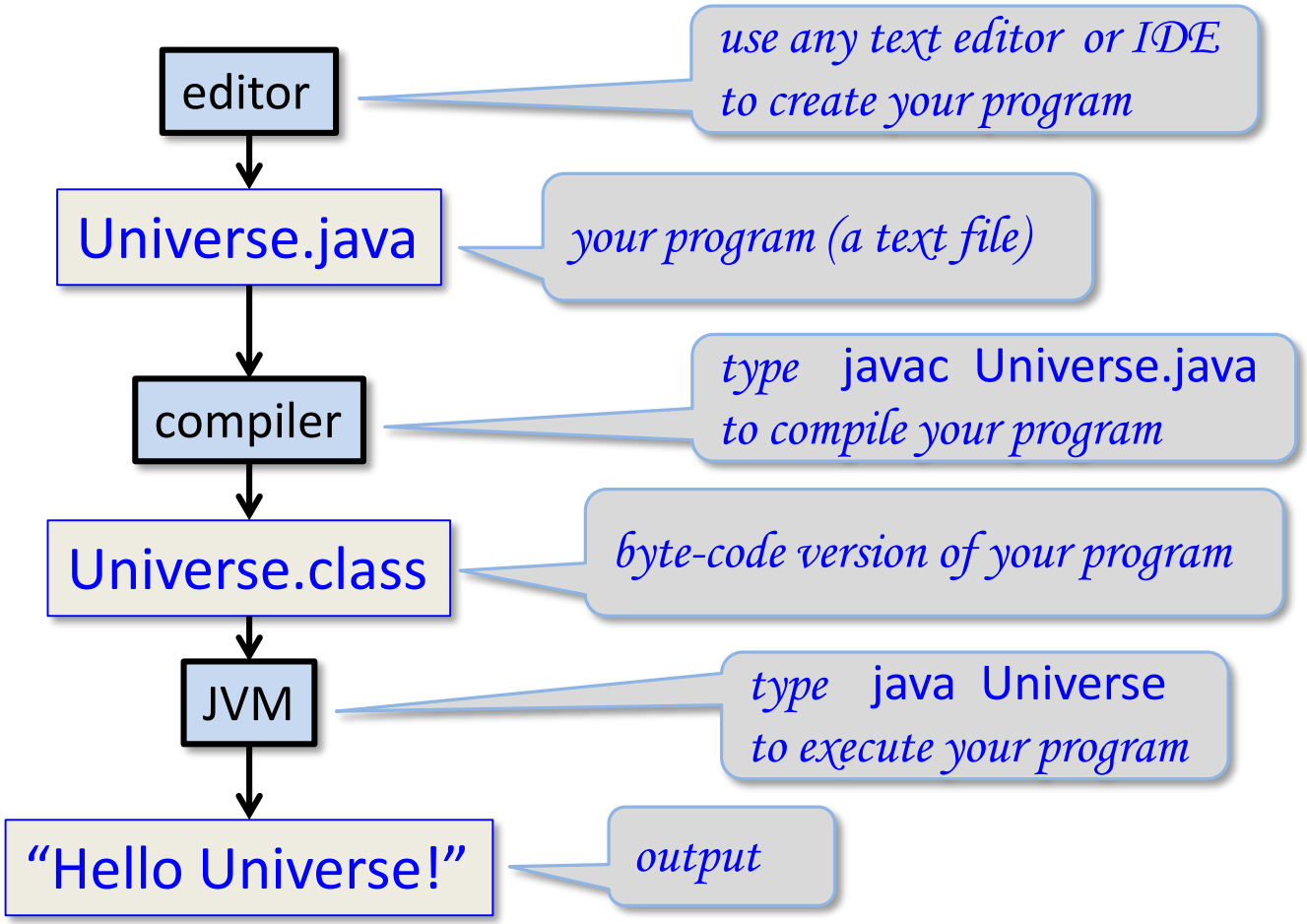
Why Java?

- Java is widely available with large online support.
- Embraces a full set of modern abstractions.
- Java's motto: write-once-run-anywhere.
- Has variety of automatic checks for mistakes in programs, especially when used in an IDE, e.g., Eclipse.
  So, it is suitable for learning to program.
- EECS2101 uses Java but is **not** a Java course.
  It is an introductory course on data structures.

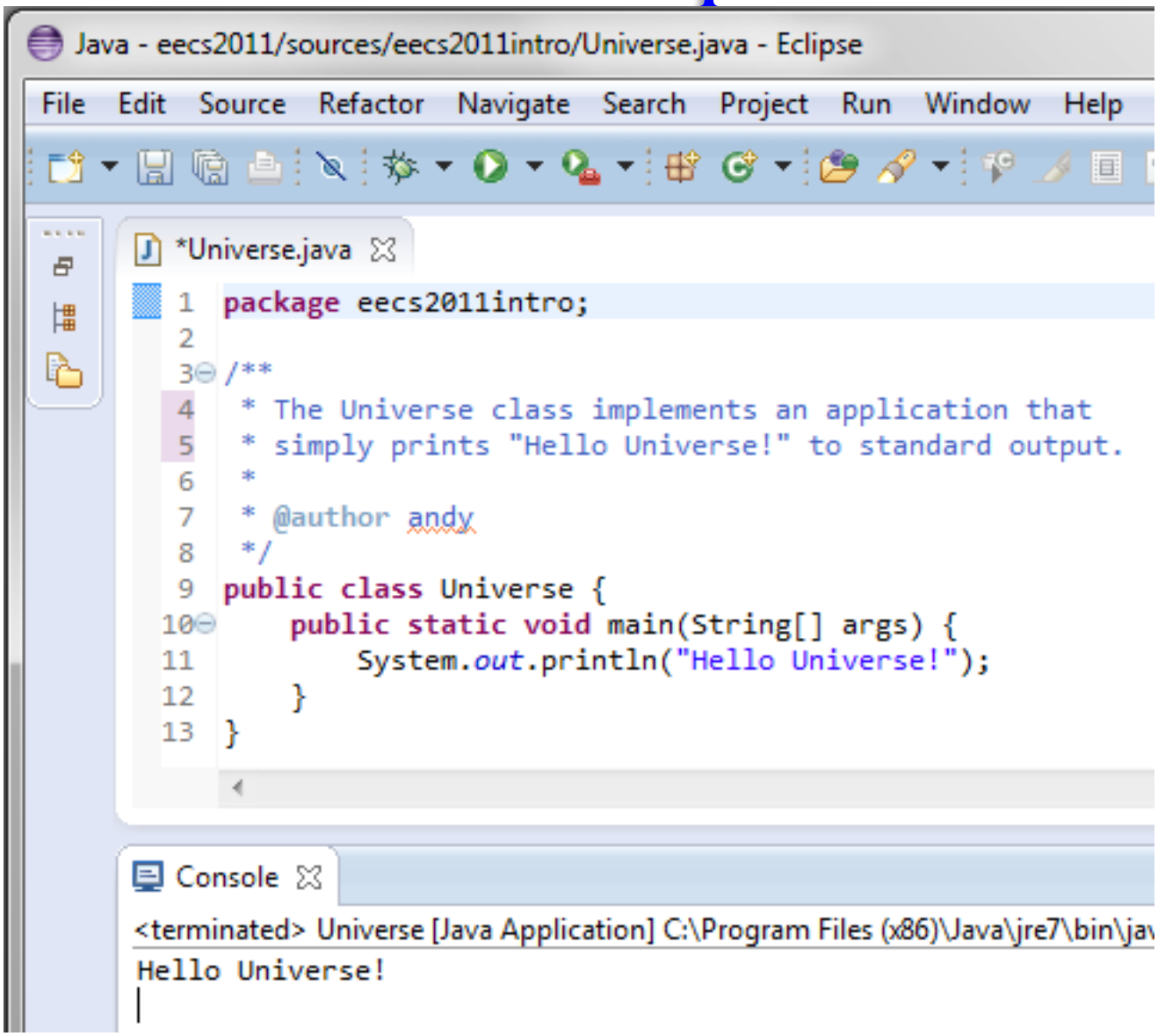
An Example Program



Developing a Java Program



An Eclipse view



program

output

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.

Reserved Words

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		

Identifiers

- An **identifier** is the name of a class, method, or variable.
- An identifier cannot be any reserved word.
- It can be any string of Java letters, beginning with a letter (or '$' or '_'), followed by letters, including digits.

Examples:

- packages: org.omg.CORBA , java.lang , eecs2011.assignments
- Classes: Name , Math , enum , MyClass , YourClass
- Interfaces: Comparable, Clonable, Iterable
- methods: verb , getValue , isEmpty , equals , clone , toString
- variables: name , counter3 , $testVariable , _testVar
- CONSTANTS: PI , MAX_INT , ELECTRON_MASS

Primitive Types

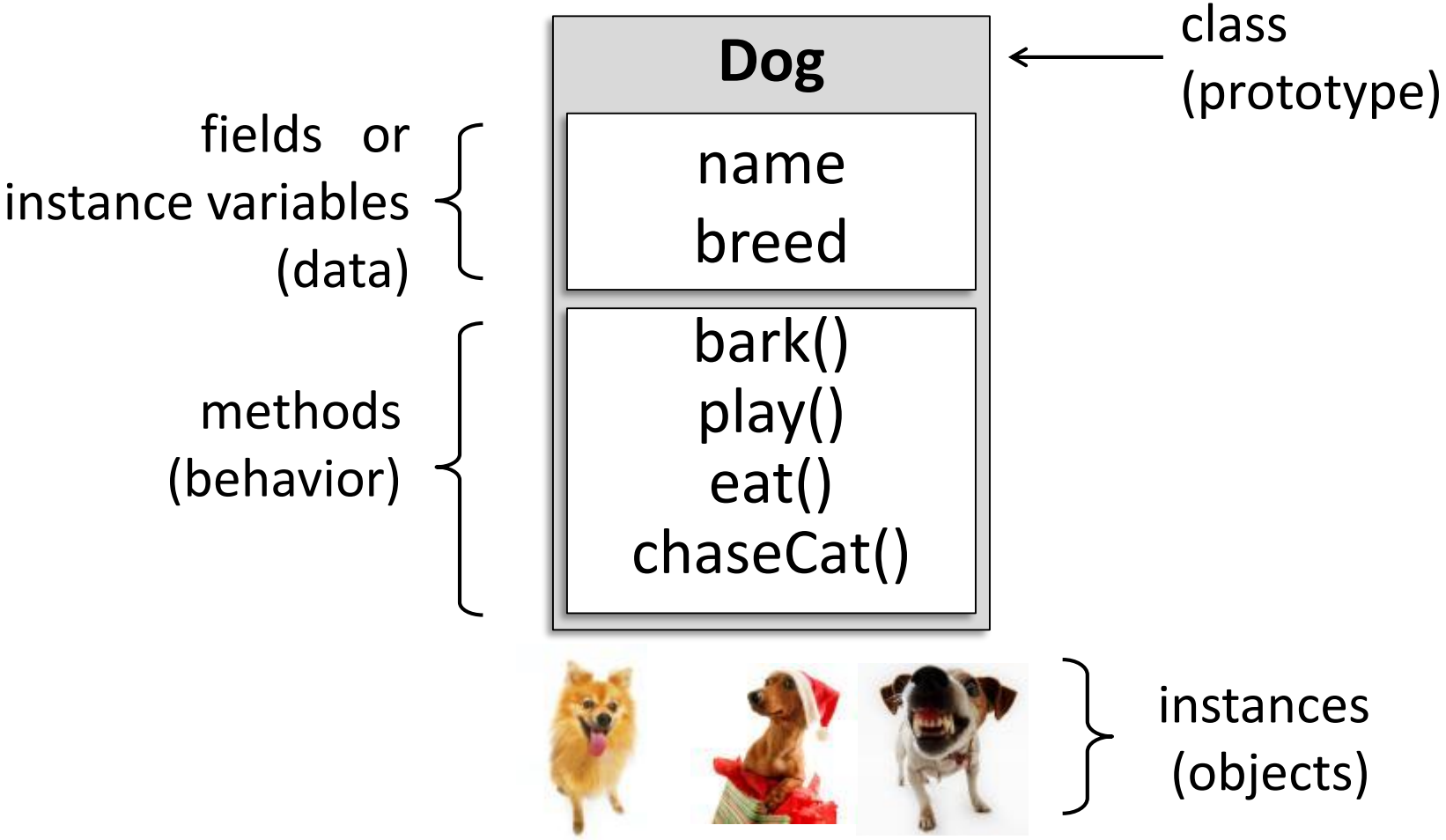
- Java has several primitive types, which are basic ways of storing data.

- An identifier variable can be declared to hold any primitive 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



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** (or **fields**) represent the data associated with an object of a class. Instance variables must have a type, which can either be a primitive 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

```
1. public class Counter {
2.   private int count;                          // a simple integer instance variable
3.   public Counter( ) { }                      // default constructor (count is 0)
4.   public Counter(int initial) { count = initial; }     // an alternate constructor
5.   public int getCount( ) { return count; }             // an accessor method
6.   public void increment( ) { count++; }                // an update method
7.   public void increment(int delta) { count += delta; } // an update method
8.   public void reset( ) { count = 0; }                  // an update method
9. }
```

- This class includes one instance variable, named **count**, with default value of zero, unless otherwise initialized.

- The class includes 2 special methods known as **constructors**, 1 accessor method, and 3 update methods.

Reference Objects

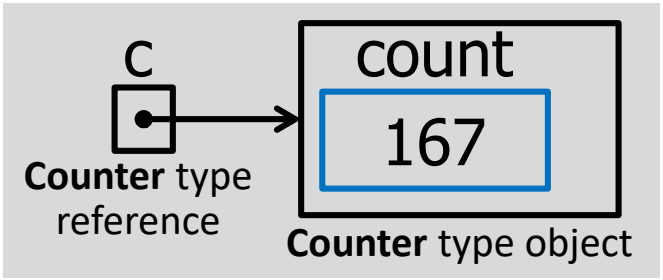
- Classes are known as **reference types** in Java, and a variable of that type is known as a **reference variable**.
- A reference variable v can be viewed as a "pointer" to some object of its class type.
- A reference variable is capable of storing the location (i.e., memory address) of an object from 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.

Creating and Using Objects

- 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 an object of that class for further use.

Example: Counter c = **new Counter**(167);



Continued Example

```
1.  public class CounterDemo {
2.    public static void main(String[ ] args) {
3.      Counter c;                      // declares a variable; no counter yet constructed
4.      c = new Counter( );             // constructs a counter; assigns its reference to c
5.      c.increment( );                 // increases its value by one
6.      c.increment(3);                 // increases its value by three more
7.      int temp = c.getCount( );       // will be 4
8.      c.reset( );                     // value becomes 0
9.      Counter d = new Counter(5);// declares and constructs a counter having value 5
10.     d.increment( );                 // value becomes 6
11.     Counter e = d;                  // assigns e to reference the same object as d
12.     temp = e.getCount( );           // will be 6 (as e and d reference the same counter)
13.     e.increment(2);                 // value of e (also known as d) becomes 8
14.   }
15. }
```

- 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

- A reference variable name of an object can be used to access the members of the object's class.
- This access is performed with the dot (".") operator.
- Such access is done by using the object reference variable name, followed by the dot operator, followed by the member field or member method name and its parameters:

Examples:

```
myObject.field
myObject.myMethod ( parameters )
e.increment ( 3 );                  // increments Counter e by 3
System.out.println (  e.getCount() );   // print counter value of Counter e
System.out.println ( e.count );     // Ooops! count field was private
```

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 the 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.

Defining Classes

- A **class definition** is a block of code, delimited by braces "**{**" and "**}**" , within which is included:
  - declarations of instance variables
  - member methods & constructors
  - nested classes

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

Modifiers

- **Access modifiers:** public, protected, private, default
- **Change modifier:** final
- **Instantiation modifier:** static
- **Field modifiers:** transient , volatile
- **Method modifiers:** abstract, native, synchronized

Access Control Modifier

Specifies who has access to a defined aspect of a class:

> ***Motto: information hiding***
>
> *make each class or member as inaccessible as possible.*

- **Private:** accessible by members of the given class.
- ***default (package-protected):*** accessible by classes in the same package as the given class.
- **protected:** accessible by subclasses, as well as classes in the same package as the given class.
- **public:** accessible by all classes.

Static Modifier

- When a field or method member of a class is declared as **static**, it is associated with the class as a whole, and is shared by all instances of that class, e.g., *instanceCount*.

- static member access: *ClassName.memberName*
  e.g., *Math.sqrt(2)* (in the Java Library package java.lang)

- Primary purpose for static vs. (non-static) instance methods:
  - **static methods:** to implement (utility) functions.
  - **instance methods:** to implement data-type operations.

Other Modifiers

- **abstract**
  - **method:** has declaration, but no body
  - **class:** can have abstract methods
  - **interface:** is abstract class with only abstract methods; no fields, no constructors.
- **final**
  - **field:** non-resettable (constant)
  - **method:** non-overridable
  - **class:** cannot be abstract or sub-classed.
- Other modifiers not discussed in this course:
  native, synchronized, transient, volatile, strictfp

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 that reference is copied (but not the object it points to).

The Keyword *this*

Within the body of a method, the keyword **this** is automatically defined as an "alias" to the instance upon which the method was invoked. There are three common uses:

- 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.
- To differentiate between an instance variable and a local variable or parameter with the same name.
- To allow one constructor body to invoke another constructor body.

Example: Line 4 on page 16 can be written as:

```
public Counter(int count) { this.count = count; }
```

Expressions and Operators

- Existing values can be combined into expressions using special symbols and keywords known as operators.

- The semantics of an operator depends upon the type of its operands (see "overloading" later).

Example:

```
int  x  =  a + b ;                    // add integers a and b
String s = "rewar" + "ding" ;   // concatenate strings
```

Arithmetic Operators

- Java supports the following **binary** 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 **double**, the result is a **double**.
- Integer division has its result truncated.

Increment and Decrement Ops

- Java provides the plus-one increment (++) and decrement (−−) **unary** operators.
  - If such an operator is used as **prefix** to a variable reference, then 1 is added to (or subtracted from) the variable, **then** its value is read into the expression.
  - If it is used as **postfix** to 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 values, 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 determined based on the value of the first operand.

Conditional Operator

- Java supports the following **ternary** operator:

(booleanExpression) ? valueIfTrue : valueIfFalse ;

// acts like if-then-else.

Example:

```
/** this method returns larger of a or b */
public static  int  max ( int a , int b ) {
        return ( a > b) ?  a  :  b;
}
```

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			Association
	Type	**Symbols**	
1	array index method call dot operator	[ ] ( ) .	Left-to-Right
2	postfix ops prefix ops cast	*exp*++ *exp*−− ++*exp* −−*exp* +*exp* −*exp* ~*exp* !*exp* (*type*) *exp*	Right-to-Left
3	mult./div.	* / %	Left-to-Right
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*	Right-to-Left
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**
  - **implicit casting**.

Explicit Casting

- Syntax: *(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.

Example:

```
double  ap = 14.696;                    // atmospheric pressure in psi
int  round = (int) (ap + 0.5);          // round to nearest integer
double apRound = (double) round;        // cast back as double. Now 15.0
```

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.

Example:

```
int  x = 2  ;     double  y = 20.0 ;
y  =  (3*y+4) / (x+2) ;        //  implicit widening cast.  Now  y == 16.0
x  =  (3*y+4) / (x+2) ;        //  Is now  x == 13  ?  No. Compiler error!  Why?
```

Quiz

```
// Broken comparator - can you spot the flaw!
public int compare ( Integer first , Integer second ) {
    return first < second ? -1 : ( first == second ? 0 : 1 );
}
```

```
// What would this return: –1 , 0, or 1 ?
compare( new Integer(12) , new Integer(12) ) ;
```

Part 1: Summary

- Java source (.java) text file & compiled (.class) byte-code file
- Reserved words & identifiers
- Primitive types & reference types
- Classes & objects
- Class members: fields, constructors, methods, nested classes
- Class & method signatures
- Keyword **this**
- **new** & dot (**.**) operators
- Wrapper types
- Modifiers: access, static, abstract, final
- Expressions & operators
- Operator precedence
- Explicit & implicit casting

Java Primer: Part 2
I/O Methods and Control Flow

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 immediately terminate a while or for loop when executed within its body.
  - ***break*** *label* *// label is optional*

- Java also supports a **continue** statement that causes the current iteration of a loop body to stop, but with subsequent passes of the loop proceeding 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 or block, and the body
  - any of these four sections 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 : collection)
    loopBody
```

For-Each Loop Example

- Computing 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.

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 Output

- Java provides a built-in static object, called **System.out**, that performs output to the "standard output" device, with 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.

Example: Input/Output

```
import java.util.Scanner ;
import java.io.PrintStream ;

public class IOExample {
    public static void main(String[ ] args) {
        Scanner input = new Scanner(System.in) ;
        PrintStream output = System.out ;
        output.print("Enter your age in years: ") ;
        double age = input.nextDouble( ) ;
        output.print("Enter your maximum heart rate: ") ;
        double rate = input.nextDouble( ) ;
        double fb = (rate – age) * 0.65 ;
        output.println("Your ideal fat-burning heart rate is " + fb) ;
        input.close( ) ;        // potential memory leak if not closed!
    }
}
```

Example Program

```
public class Newton {
    public static double sqrt ( double a ) {
        final double EPSILON = 1e-15;      // relative error tolerance
        if ( a < 0.0 ) return Double.NaN ;
        // compute approximate root of f(x) = x^2 – a by Newton's method:
        double x = a;                      // initialize iteration variable
        while ( Math.abs (x – a/x) > EPSILON * x )
            x = (x + a/x) / 2;             // Newton iterate: x = x – f(x)/f'(x)
        return x;                          //   (1 − ε)√a ≤ x ≤ (1 + ε)√a
    }

    public static void main( String[] args) {
        System.out.println ( "Input numbers & their square roots:") ;
        double root ;
        for (int i = 0; i < args.length; i++ ) {
            root = sqrt ( Double.parseDouble( args[i] ) ) ;
            System.out.println( args[i] + " \t " + root ) ;
        }
    }
}
```

Output:

```
Input numbers & their square roots:
-5          NaN
Infinity    Infinity
0           0.0
1           1.0
2           1.414213562373095
3           1.7320508075688772
4           2.0
```

Another 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   }
```

Program cont'd

```
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  }
```

Program cont'd

```
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  }
```

Packages & imports

- /* name of package that contains the class definition goes on the first line */
- **import** packageName; // appears next
- **import** java.util.Scanner;
- **import** java.util.*;
- /* **static import** as of Java 5
  e.g., importing a class with static members */

```
import static packageName.PhysicalConstants.*;
import static java.lang.Math.*;
    // now you can say sqrt() instead of Math.sqrt()
```

Coding

- Algorithm pseudo-code facilitates

  feasibility , efficiency & verification analysis

  prior to time consuming & costly code development

- Packages & imports
  - enhanced modularity & encapsulation
  - avoids naming conflicts
- IDEs , e.g. Eclipse
- Documentation & Style — Javadoc

Errors

- Compile-time **int** x = 1.5;
- Run-time x/0;
- Logical errors bugs are the bane of programmer's existence!

> ***Motto: early detection***
> *catch mistakes as early as possible; preferably at compile time.*

- **Example:** An "expensive" bug
  - *Ariane 5* European satellite
  - Cost: $7B, 10 years to build
  - Rocket crashed within 1 min of takeoff with expensive uninsured scientific cargo
  - bug: unintended cast from 64-bit to 16-bit.

Testing & Debugging

- Top-down modular testing with stubs
- Bottom-up unit-testing
- Regression testing on evolving software
- Basic debugging by print statements or assertions:

```
assert ( x > 5) : "x = " + x ;
// if false, throw an AssertionError
```

- Enabling & disabling assertions:

```
java –ea    // enable assertions. Slow – used during testing
java –da    // disable assertions. Fast – for production
```

- Java debugger jdb (using breakpoints)
- Advanced IDE debuggers with GUI display

Part 2: Summary

- Control Flow Statements:
  - Branching: if-then-else, switch, break, continue
  - Loops: for-loop, while-loop, do-while, for-each
- Simple I/O
- Example programs
- Packages & imports
- Coding
- Errors
- Testing & Debugging