Java Basics

Structure of a Java Program

A program consists of one or more class definitions

```
class NameOfClass
{
        // body of the class
}
```

A class definition consists of:

- Variable declarations

```
int num, k = 0, sum;
double x, y = 0.0;
boolean p = false, q;
```

- Constant declarations

```
static final int MAGIC = 666;
static final double E = 2.71;
```

- Method declarations

```
static boolean isPrime (int p)
{   // body of isPrime   }

void process (double x, double y)
{   // body of process   }

String convert ()
{   // body of convert   }

int getValue ()
{   // body of getValue   }
```

These declarations may have visibility modifiers:

public

protected

private

Other modifiers on methods:

static	**abstract**	**synchronized**
final	**native**	

Primitive Data Types (8)

boolean

char

byte, **short**, **int**, **long**

float, **double**

All eight primitive data types have required sizes and default values.

Type: **boolean**

Values:	true, false
Default:	false
Size:	1 byte

Type: **char**

Values:	Unicode character
Default:	\u0000
Size:	16 bits
Minimum:	\u0000
Maximum:	\uFFFF

Type: **byte**

Values:	signed integer
Default:	0
Size:	8 bits
Minimum:	-128
Maximum:	127

Type: short

Values:	signed integer
Default:	0
Size:	16 bits
Minimum:	-32,768
Maximum:	32,767

Type: int

Values:	signed integer
Default:	0
Size:	32 bits
Minimum:	-2,147,483,648
Maximum:	2,147,483,647

Type: long

Values:	signed integer
Default:	0
Size:	64 bits
Minimum:	-9,223,372,036,854,775,808
Maximum:	9,223,372,036,854,775,807

Type: float

Values:	IEEE 754 floating-point
Default:	0.0
Size:	32 bits
Minimum:	±3.40282347E+38
Maximum:	±1.40239846E-45

Type: double

Values:	IEEE 754 floating-point
Default:	0.0
Size:	64 bits
Minimum:	±1.79769313486231570E+308
Maximum:	±4.94065645841246544E-324

All other data are types of objects.

Java Applications

Stand-alone programs that run under control of the operating system:

- May accept input and produce output at the terminal.
- May read and write files.
- May build graphical elements.

One class in program is **public**, and it contains a method definition of the form:

```
        public static void main(String [ ] args)
        {
                // body of the method
        }
```

- Execution always begins by invoking this main method.
- The public class name must be same as file name

    class Abundant resides in the file Abundant.java

- Note: Java is case sensitive.

Example Application: Abundant Number

```
public class Abundant
{
    public static void main (String [] args)
    {
        int sum, div, quot;
        int num = 3;
        do
        {
            num = num+2;
            sum = 1;
            for (div=3; div <= Math.sqrt(num); div=div+2)
            {
                quot = num/div;
                if (num == div*quot)
                {
                    sum = sum+div;
                    if (div != quot)  sum = sum+quot;
                }
            }
        }
        while (sum <= num);

        System.out.println("First odd abundant number = " + num);
    }
}
```

Running Java Applications

Compiler

$$\begin{tikzcd} \text{Abundant.java} \arrow[r] & \boxed{\text{javac}} \arrow[r] & \text{Abundant.class} \end{tikzcd}$$

The Java program is translated into the language of the Java Virtual Machine (JVM), which is then interpreted (simulated) to produce the runtime results.

Interpreter

$$\begin{tikzcd} \text{Abundant.class} \arrow[r] & \boxed{\text{java}} \arrow[r] & \text{runtime results} \end{tikzcd}$$

On a Unix (Linux) System

If the source is in the file Abundant.java, type

```
javac Abundant.java
```

to compile and create a target file calledAbundant.class in the current directory.

To execute the program, type

```
java Abundant
```

Note: If the source file has more than one class, as many target files will be created.

Execute the (public) class containing the main method.

Object-Oriented Programming (OOP)

An object is an entity that encapsulates

- Attributes in the form of data
- Behavior in the form of methods.

In addition, an object has an identity that distinguishes it from all other objects.

Objects have the ability to hide their internal make-up, presenting a well-defined but restricted public interface of operations (methods) to users of the object.

Normally, objects are manipulated by calling their methods, which are executed by an object, possibly changing the state of the object.

Calling a method of an object is also called sending a message to the object.

Objects are defined by classes that specify the data and methods that belong to the objects of the class.

An objects is referred to by a variable whose type is specified by the class that defined the object.

Classes

A class definition defines a type of objects.

- Blue-print for stamping out object of the new type.
- Factory that makes objects of a certain form.
- Contains local data (state of the objects) and methods (behavior of the objects).

Example

```
class Domino
{
        data declarations
                and
        method definitions
}
```

Data declarations take two forms

1. Instance Variables

   Every object defined by the class contains
   its own copy or instance of these variables.

   Each object may store different values in these variables.

2. Class Variables (have **static** modifier)

   These variables belong to the class itself and are shared
   by all of the objects fabricated from the class.

Methods take three forms

1. Instance methods

   These methods operate directly on the state of an
   object in the form of its instance variables.

   Referenced by specifying an object and a method.

   ```
   objectName.methodName(parameters)
   ```

2. Constructors

   These instance methods share their name with the class.

   Have no specification of return type.

   Parameters may be provided as initialization information.

   Automatically invoked when Objects are created using **new**.

3. Class Methods (have **static** modifier)

   Belong to the class itself and are shared by all of the objects of the class.

   Cannot refer directly to instance variables and instance methods.

   Called using the class name.

   ```
   className.methodName(parameters)
   ```

   For example,

   ```
   Math.sqrt(99.9)
   ```

Creating Objects

Suppose we have a class called Domino (a domino factory).

Variable Declaration

```
Domino d,e;             // Declares two variables that
                        // may refer to Domino objects,
                        // but are null initially.
```

Note: Declarations do not create objects (unlike C++).

Instance Creation

```
d = new Domino();
                // new creates a Domino object and
                // automatically calls a constructor for
                // Domino (maybe default constructor).
```

Declaration and Creation Combined

```
Domino d2 = new Domino(5, 8, true);
```

Definition of Domino Class

```
class Domino
{
// Instance Variables — usually private
      int spots1;
      int spots2;
      boolean faceUp;

// Class Variables
      static final int MAXSPOTS = 9;             // a constant
      static int numDominoes = 0;

// Constructors (sometimes include validation)
      Domino(int val1, int val2, boolean up)
      {
            if (0<=val1 && val1<=MAXSPOTS)
                spots1 = val1;
            else spots1 = 0;
            if (0<=val2 && val2<=MAXSPOTS)
                spots2 = val2;
            else spots2 = 0;
            faceUp = up;
            numDominoes++;
      }

      Domino()              // overloading
      {
            this(0, 0, false);                  // Calls constructor with
      }                                         // matching parameters.
// Instance Methods
    int getHigh()
    {
        if (spots1 >= spots2) return spots1;
        else return spots2;
    }

    int getLow()
    {
        if (spots1 <= spots2) return spots1;
        else return spots2;
    }

    public String toString()
    {
        String orientation = faceUp ? "UP" : "DOWN";
        return "<" + getLow() + ", " + getHigh() + ">  " + orientation;
    }

    boolean matches(Domino otherDomino)
    {
        int a = otherDomino.getHigh();
        int b = otherDomino.getLow();
        int x = getHigh();
        int y = getLow();
        return a==x || a==y || b==x || b==y;
    }

// Class Methods
    static int getNumber()
    {
        return numDominoes;
    }
}   // end of Domino class
```

Note: UVariable declarations and method definitions may come in any order in a class, without the need for prototypes as in C or C++.

Application Class that Uses Domino

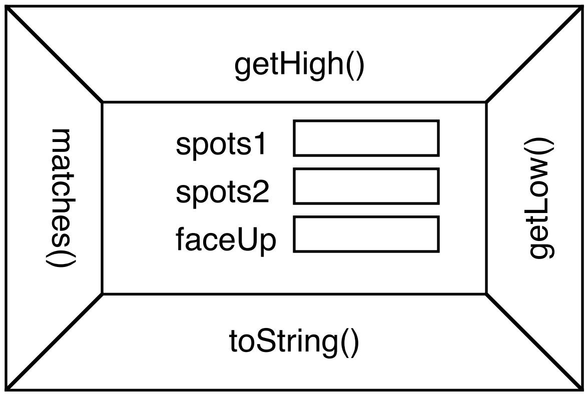
```
public class Dominoes
{
    public static void main(String [] args)
    {
        Domino d1, d2, d3, d4;
        d1 = new Domino(3, 5, true);
        d2 = new Domino(4, 4, false);
        d3 = new Domino();
        d4 = new Domino(8, 5, false);

        System.out.println("Domino 1: " + d1);
        System.out.println("Domino 2: " + d2);
        System.out.println("Domino 3: " + d3);
        System.out.println("Domino 4: " + d4);

        System.out.println("Matches 1 and 2: " + d1.matches(d2));
        System.out.println("Matches 1 and 4: " + d1.matches(d4));
        System.out.println("Number of Dominoes: "
                                        + Domino.getNumber());
    }
}   // end of Dominoes class
```

Output from executing Dominoes

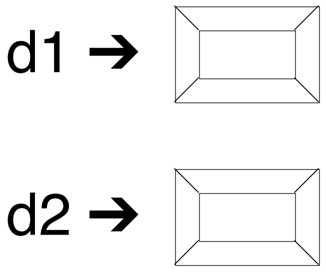
```
Domino 1: <3, 5>  UP
Domino 2: <4, 4>  DOWN
Domino 3: <0, 0>  DOWN
Domino 4: <5, 8>  DOWN
Matches 1 and 2: false
Matches 1 and 4: true
Number of Dominoes: 4
```

Visualizing Objects

Domino objects defined by the class Domino can be viewed as capsules:



The declaration of Domino variables and objects can be viewed as references to the objects:



Note that object variables follow pointer semantics.

After the assignment

```
        d1 = d2;
```

both variables refer to the same object.

Kinds of Classes

1. Named Collections of data declarations
   - Normally have only instance variables.
   - Called records in Pascal and Ada, and structs in C and C++.
   - Class is instantiated to create variables that refer to these objects.
   - Seldom used in Java.

Example

```
class StudentRec
{
     String name;
     int classification;        // 1, 2, 3, 4
     double gpa;
}
```

2. Named Collections of methods
   - Normally only class (**static**) methods.
   - Some constants (also **static**) may occur.
   - These classes are *not* instantiated—they have no objects.
   - Examples in Java: System and Math.

Example

```
public final class java.lang.Math
                                          extends java.lang.Object
{
// Fields
     public final static double E;
     public final static double PI;

// Methods
     public static double sqrt (double a);
     public static double pow (double a, double b);
     public static double random ();
     public static int abs (int a);
     public static long abs (long a);
     public static float abs (float a);
     public static double abs (double a);
     public static double ceil (double a);
     public static double floor (double a);
     public static int round (float a);
     public static long round (double a);
     public static double rint (double a);
```

```
        public static double IEEEremainder (double f1, double f2);
        public static double log (double a);
        public static double exp (double a);
        public static int max (int a, int b);
        public static long max (long a, long b);
        public static float max (float a, float b);
        public static double max (double a, double b);
        public static int min (int a, int b);
        public static long min (long a, long b);
        public static float min (float a, float b);
        public static double min (double a, double b);
        public static double sin (double a);
        public static double cos (double a);
        public static double tan (double a);
        public static double asin (double a);
        public static double acos (double a);
        public static double atan (double a);
        public static double atan2 (double a, double b);
}
```

3. Classes that define a new sort of Object
    - Instance variables to maintain the state of the objects.
    - Instance methods to access and manipulate data in the objects.
    - Possibly some class variables and methods.
    - Normally must be instantiated to be used
      (need to create objects of the new sort).
    - True object-oriented programming.

Example

Domino

Domino objects have

State: spot1, spot2, faceUp

Behavior: getHigh(), getLow(), toString(), matches(d).

Domino objects need to be created (instantiated) using the **new** operator.

Another Constructor for Domino

```
Domino(boolean up)
{
	spots1 = (int)((MAXSPOTS + 1)*Math.random());
	spots2 = (int)((MAXSPOTS + 1)*Math.random());
	faceUp = up;
	numDominoes++;
}
```

Create Some Random Dominoes

```
	Domino dx = new Domino(true);
	Domino dy = new Domino(false);

	System.out.println(dx);
	System.out.println(dy);
	System.out.println(dx.matches(dy));
```

Possible Output:

```
<2, 7> UP
<0, 8> DOWN
false
```

Errors

What happens if we call

```
Integer.parseInt("12e4")?
```

The string cannot be parsed as an integer, so a runtime error occurs:

```
java.lang.NumberFormatException: 12e4
```

In Java, runtime errors are called exceptions.

When such an error occurs, we say an exception has been thrown.

Java has an extensive facility for handling exceptions, which are themselves objects.

Example

NumberFormatException is an exception thrown by:

```
Integer(String s);
Integer.parseInt(String s);
Integer.valueOf(String s);
```

Java Reserved Words

abstract	final	public
assert	finally	return
boolean	float	short
break	for	static
byte	if	strictfp
case	implements	super
catch	import	switch
char	instanceof	synchronized
class	int	this
continue	interface	throw
default	long	throws
do	native	transient
double	new	true
else	null	try

enum	package	void
extends	private	volatile
false	protected	while

Reserved but not used

const	goto

Simple Output

IO is performed on objects called streams.

The standard IO streams are found in System.

```
public final class  java.lang.System extends java.lang.Object
{
// Fields
        public static InputStream in;
        public static PrintStream out;
        public static PrintStream err;

// Methods
            :
}
```

The predefined objects, in and out, provide input from the keyboard and output to the display screen.

Output to the screen is performed using the instance methods *print* and *println* for a PrintStream.

System.out is a PrintStream object.

```
System.out.println("x = " + x);    // puts a newline after output
System.out.print("y = " + y);      // just prints the output
```

import

An import clause does not ask that certain classes be loaded, but only that they be made visible.

The Java runtime system automatically loads classes when they are first accessed in a program.

By importing the classes in a package, those classes may be named directly and do not require a full package specification.

Examples

- **import** java.util.ArrayList:

  Makes the ArrayList class in the *java.util* package visible.

  Without the import we would have to write java.util.ArrayList every time we use the class.

- **import** java.util.*:

  Makes the all the classes and interfacess in the *java.util* package visible.

Note that the classes in the package *java.lang* are always visible in every Java program.

Expressions

Operators together with elementary expressions, such as variables, literals, and method applications, form the expressions of Java.

Expressions are those language constructs that produce values.

The other components of languages are declarations and commands.

Elementary expressions

- Simple variables: sum, x , domino2 (value of domino2 is a reference to an object).
- Instance and class variables: d2.spots1

- Array variables: a[2], a[k]
- Method applications: d1.getLow(), Math.random(), Integer.parseInt(str).

Precedence

- Elementary expressions (variables and method calls) have highest precendence.
- Parentheses may always be used to force the precedence.
- Java has thirteen levels of precedence for its unary, binary, and ternary operators.

Note: Some operators cause *side effects*—changes in values of the operands: k++

Java Operators

Type Conventions

integer	**int** (**byte**, **short**) and **long**
number	integer, **float**, and **double**
primitive	number, **char**, and **boolean**
Object	any object
String	any String object
Class	any Class (constructor)
type	any type name
any	any type

Precedence	Operator Signature	Associativity
1	++ : numberVariable → number	none
1	-- : numberVariable → number	none
1	+ : number → number	right
1	- : number → number	right
1	~ : integer → integer	right
1	! : **boolean** → **boolean**	right
1	(*type*) : any → any	right

2	*	: number, number → number	left
2	/	: number, number → number	left
2	%	: number, number → number	left
3	+	: number, number → number	left
3	-	: number, number → number	left
3	+	: String, String → String	left
4	<<	: integer, integer → integer	left
4	>>	: integer, integer → integer	left
4	>>>	: integer, integer → integer	left
5	<	: number, number → **boolean**	none
5	<=	: number, number → **boolean**	none
5	>	: number, number → **boolean**	none
5	>=	: number, number → **boolean**	none
5	**instanceof**	: Object, Class → **boolean**	none
6	==	: primitive, primitive → **boolean**	left
6	!=	: primitive, primitive → **boolean**	left
6	==	: Object, Object → **boolean**	none
6	!=	: Object, Object → **boolean**	none
7	&	: integer, integer → integer	left
7	&	: **boolean**, **boolean** → **boolean**	left
8	^	: integer, integer → integer	left
8	^	: **boolean**, **boolean** → **boolean**	left
9	|	: integer, integer → integer	left
9	|	: **boolean**, **boolean** → **boolean**	left
10	&&	: **boolean**, **boolean** → **boolean**	left
11	||	: **boolean**, **boolean** → **boolean**	left
12	? :	: **boolean**, any, any → any	right
13	=	: variable, any → any	right
13	*=	: numberVariable, number → number	right
13	/=	: numberVariable, number → number	right
13	%=	: numberVariable, number → number	right
13	+=	: numberVariable, number → number	right
13	+=	: StringVariable, String → String	right
13	-=	: numberVariable, number → number	right
13	<<=	: integerVariable, integer → integer	right
13	>>=	: integerVariable, integer → integer	right
13	>>>=	: integerVariable, integer → integer	right
13	&=	: integerVariable, integer → integer	right
13	^=	: integerVariable, integer → integer	right
13	|=	: integerVariable, integer → integer	right
13	&=	: booleanVariable, **boolean** → **boolean**	right
13	^=	: booleanVariable, **boolean** → **boolean**	right
13	|=	: booleanVariable, **boolean** → **boolean**	right

Coercion and Conversion (casting)

Mechanisms that convert a value of one data type into a "corresponding" value of another type.

Conversion

- Transformation done by an explicit operator.

  long Math.round(**double** a);

  Math.round(8.72) returns the **long** 9

- Casting (explicit casting)

  double d = 44.84;

  int m = (**int**)d;   // truncate

Coercion

- Automatic transformation code generated by the compiler.
- Casting (implicit casting)

  int n = 552;

  double d = n;

- Here an **int** is "impersonating" a **double**.

Where Casting is an Issue

Assume the declaration

int k = 13;

1. Assignment:

   long g = k;

2. Parameter passing:

   static void meth(**long** g) { … }

   meth(k);   // method call

3. Operands in an expression:

   long g = 12345678900L;   // L means long

   long h = k + g;

Note: **int** m = k + g; is illegal.
Use: **int** m = (**int**)(k + g);

Coercion and Conversion

Lossless

- Widening: Convert value to a “larger” or more general type.
- Called a promotion.
- Coercion in Java (implicit).
- Promotions in Java

$$\begin{tikzcd}
\textbf{byte} \arrow[r] & \textbf{short} \arrow[r] & \textbf{int} \arrow[r] & \textbf{long} \arrow[r] & \textbf{float} \arrow[r] & \textbf{double} \\
 & & \textbf{char} \arrow[u] & & &
\end{tikzcd}$$

Lossy

- Narrowing: Potentially information may be lost.
- Explicit cast or conversion in Java.

  double d = 25.75;
  byte b = (**byte**)d; // truncate

Table legend

Consider an assignment command,

<variable> = <expression> ;

- Column on left shows the type of <variable>.
- Top row shows the type of <expression>.

	char	**byte**	**short**	**int**	**long**	**float**	**double**	**boolean**
char	same	convert	convert	convert	convert	convert	convert	none
byte	convert	same	convert	convert	convert	convert	convert	none
short	convert	coerce	same	convert	convert	convert	convert	none
int	coerce	coerce	coerce	same	convert	convert	convert	none
long	coerce	coerce	coerce	coerce	same	convert	convert	none
float	coerce	coerce	coerce	coerce	coerce	same	convert	none
double	coerce	coerce	coerce	coerce	coerce	coerce	same	none
boolean	none	none	none	none	none	none	none	same

Example

```
class Promo
{
	public static void main(String [] a)
	{
		long g = 123456789012345678L;
		System.out.println("g = " + g);
		float f;
		f = g;
		System.out.println("f = " + f);
		g = (long)f;
		System.out.println("g = " + g);
	}
}
```

Output

```
	g = 123456789012345678
	f = 1.234567e+17
	g = 123456790519087104
```

Numeric Literals

Integer literals, such as 25404 and -1109, are normally interpreted as of type **int**.

Literals of type **long** can be written using an L after the numeral:

```
25404L          111222333444555666L
```

The declaration

```
long x = 12345678900;
          ^
```

produces the error message: Numeric overflow.

Floating-point literals, such as 3.1415926, are interpreted as of type **double**.

Literals of type **float** can be written using an F:

```
3.1415926F          0.001F
```

Note: Lower-case specifications (l and f) are allowed but are not as readable.

An Exception

In variable declarations and assignment, **byte** and **short** variables may be given literal initial values by writing integers of the appropriate size:

```
byte b = -25;
short s = 10101;
```

These variables may be assigned new values, provided those values are within the ranges of **byte** and **short**, respectively.

But, these apparent coercions of **int** to **byte** and **int** to **short** are not allowed with parameter passing.

```
void calculate(byte b)
{ … }
```

This method cannot be called using: calculate(13);

The required call is: calculate((**byte**)13);

Kinds of Identifiers

1. Module (collection of definitions) identifiers
   - Class
   - Package
   - Interface (“stripped-down” class)
2. Method identifiers
   - Class methods
   - Instance methods (including constructors)
3. Variable identifiers
   - Class variables
   - Instance variables
   - Local variables
   - Method parameters (constructor parameters)
   - Exception handler parameters

Scope (Visibility) of Identifiers

1. **Class scope** (at least)
   - The entire class

     Class methods

     Instance methods (and constructors)

     Class variables (including constants)

     Instance variables

   - Order of these declarations makes no difference (unlike C/C++) with one exception.

     int m = n;
     int n = 5;   // is illegal (also for **static**)

   - These variables are automatically initialized to default values unless they are **final** (constants).

2. **Block scope** (method scope)

   - Method parameters: Entire method.

- Local variables: Point of declaration to the end of the block (method).

  Block command:

  ```
  { // declarations and commands }
  ```

- Local variable declarations may hide class and instance variables, but not other local variables: local variables may not hidden in inner scopes.

Example

```
public class Var
{
    public static void main(String [] args)
    {
        Var v = new Var();
        Var w = new Var();
        System.out.println("v.a = " + v.a);                    // See Note
        System.out.println ("b = " + b);
        System.out.println ("c = " + c);
        {   // block command
            double a = 1.1;         // okay
            int v = 0;              // illegal
            System.out.println("a = " + a);
        }
    }
    int a;                          // instance variable
    static int b;                   // class variable
    static final int c = 99;  // constant class variable
}
```

Note

Writing   System.out.println("a = " + a);

in main is illegal. Why?

Example

```
public class Local
{
   public static void main(String [] args)
   {
```

```
        Local v = new Local();
        System.out.println ("v.strange(5) = " + v.strange(5));
        System.out.println ("v.a = " + v.a);
    }

    int strange(int b)              // hides static b
    {
        int a = 20;
        this.a = 2 * (a + b) * c + Local.b;
        System.out.println ("a = " + a);
        return 10 * this.a;
    }
    int a;                                  // instance variable
    static int b = -200;                    // class variable
    static final int c = 10;                // constant
}
```

this

1. Instance variables (and instance methods) of an object may always be accessed using **this**, which refers to the object that is currently executing the method.

2. Inside a constructor, **this** is used to invoke another constructor for the class. It must be the first command in the constructor.

Class variables (and class methods) may always be accessed using the class name.

Local Variables and Parameters

- Order of declaration my be significant.
- No automatic initialization.

For-Loop Local Variables

```
        for (int k=1; k<=100; k++) command;
```

is equivalent to

```
        {       int k;
                for (k=1; k<=100; k++) command;
        }
```

Overloading

Two or more similar but distinct operation identifiers share the same name.

Operator Overloading in Java

$+_{int,int}$, $+_{long,long}$, $+_{float,float}$, $+_{double,double}$, $+_{String,String}$,

$+_{int}$, $+_{long}$, $+_{float}$, $+_{double}$

- Similarly with other arithmetic operations.
- No user defined operator overloading in Java (unlike C++).

Method/Constructor Overloading

- Ten versions of println().
- Two versions of the constructor Integer().
- Two versions of Integer.parseInt().

Java does allow user-defined method overloading.

Overloaded operations and methods must be distinguishable by the number or types of their parameters, called the **parameter signature** of the method.

Example

```
public class Load
{
    static int fun (byte m)
    { return m+1; }

    static int fun (short m)
    { return m+10; }

    static int fun (int m)
    { return m+100; }

    static long fun (long m)
    { return m+1000; }

    static double fun (double x, int y)
    { return x+y; }

    static double fun (int x, double y)
    { return x*y; }

    // Six different operations called fun.
    // Overloaded methods may have different  return types.

    public static void main (String [] a)
    {                                                       // returns
        System.out.println (fun((byte)13));            // 14
        System.out.println (fun((short)666));  // 676
        System.out.println (fun(40000));               // 40100
        System.out.println (fun(333L));                // 1333
        System.out.println (fun(3.0, 8));              // 11.0
        System.out.println (fun(3,8.0));               // 24.0
        System.out.println (fun(3,8));                 // compile error
    }
}
```

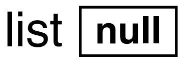
Note the conflict between overloading and coercion.
Error Message: “Reference to fun is ambiguous.”

Arrays

1. Arrays of primitive types

- Declare an array variable

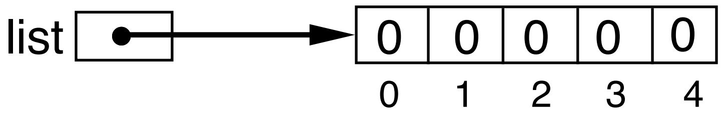
    int [] list;



    int [] a,b,c;

- Allocate an array

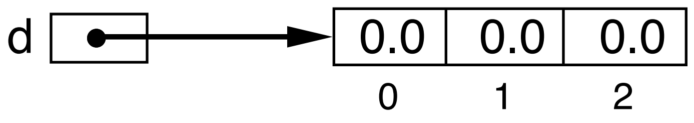
    list = **new int** [5];



Note: Automatic initialization of components.

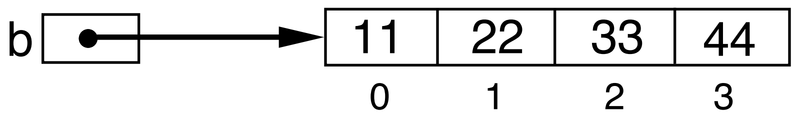
- Combined

  double [] d = **new double** [3];



- Initializers

  int [] b = { 11, 22, 33, 44 };



  With initializers, no need for **new**.

- Array Access

```
b[2]                // returns 33
b[3] = 66;   // changes a component
b[0]++;             // changes b[0]
```

- Length of an array

  Instance variable: length

```
list.length   // has value 5
d.length            // has value 3
b.length            // has value 4
```

Example

Generate 1000 random digits (0..9).
Count the occurrences of each of the digits generated.

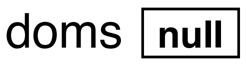
```
public class Ran
{
    public static void main (String [] args)
    {
        int [] freq = new int [10];  // 0 to 9, initialized to 0
        int m;
        for (int k = 0; k < 1000; k++)
        {
            m = (int)(Math.random()*10);
            freq[m]++;
        }
        for (int k = 0; k < 10; k++)
            System.out.println ("Number of " + k
                                    + "'s = " + freq[k]);
    }
}
```

```
Number of 0's = 92          Number of 5's = 111
Number of 1's = 80          Number of 6's = 108
Number of 2's = 94          Number of 7's = 94
Number of 3's = 109         Number of 8's = 109
Number of 4's = 108         Number of 9's = 95
```

2. Arrays of objects

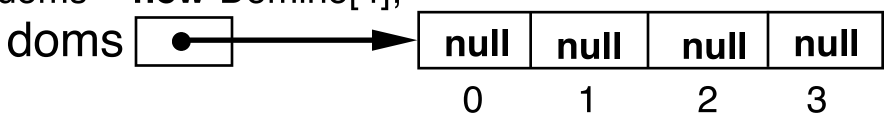
- Declare an array variable

  `Domino [] doms;`



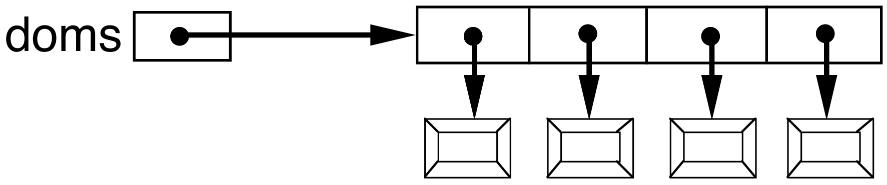
- Allocate an array

  doms = **new** Domino[4];



- Create objects for the components

```
for (int k = 0; k < doms.length; k++)
        doms [k] = new Domino();
```



Note that creating a variable that refers to an array of object takes three steps, but they can be combined.

```
Domino [] dlist =
   { new Domino(2,3,true), new Domino(7,9,true),
       new Domino(1,1,true), new Domino(4,9,true) }
```

Example

Assume that the Domino class is visible.

Create ten random dominoes and see which ones match.

```
public class Match
{
    static final int NUM = 10;
    public static void main(String [] args)
    {
        Domino [] doms;
        doms = new Domino [NUM];
        for (int k=0; k<doms.length; k++)
            doms[k] = new Domino(true);
        for (int k=0; k<doms.length-1; k++)
            for (int m=k+1; m<doms.length; m++)
            {
                if (doms[k].matches(doms[m]))
                System.out.println("Domino " + k + ": "
                     + doms[k] + " matches "
                     + "Domino " + m + ": " + doms[m]);
            }
            System.out.println("Number of Dominoes: "
                                              + Domino.getNumber());
        }
}
```

Output

```
Domino 0: <1, 7>  UP matches Domino 7: <0, 7>  UP
Domino 0: <1, 7>  UP matches Domino 8: <0, 7>  UP
Domino 0: <1, 7>  UP matches Domino 9: <0, 7>  UP
Domino 2: <4, 9>  UP matches Domino 3: <4, 9>  UP
Domino 2: <4, 9>  UP matches Domino 4: <4, 8>  UP
Domino 2: <4, 9>  UP matches Domino 5: <4, 8>  UP
Domino 2: <4, 9>  UP matches Domino 6: <4, 8>  UP
Domino 3: <4, 9>  UP matches Domino 4: <4, 8>  UP
Domino 3: <4, 9>  UP matches Domino 5: <4, 8>  UP
Domino 3: <4, 9>  UP matches Domino 6: <4, 8>  UP
Domino 4: <4, 8>  UP matches Domino 5: <4, 8>  UP
Domino 4: <4, 8>  UP matches Domino 6: <4, 8>  UP
Domino 5: <4, 8>  UP matches Domino 6: <4, 8>  UP
Domino 7: <0, 7>  UP matches Domino 8: <0, 7>  UP
Domino 7: <0, 7>  UP matches Domino 9: <0, 7>  UP
Domino 8: <0, 7>  UP matches Domino 9: <0, 7>  UP
Number of Dominoes: 10
```

Method Parameters

Method definition

```
void meth (int k, char [] ch, Domino d)
{ .... }
```

- *k*, *ch*, *d* are called formal parameters.
- Must be simple variables.
- These act as local variables in the method.

Method call

```
int num = 5;
char [] vowels = { 'a', 'e', 'i', 'o', 'u' };
Domino mine = new Domino (3, 5, true);
meth (2*num+1, vowels, mine);     // call
```

- Any expressions of the appropriate types may be passed as actual parameters.

Condition

Formal and actual parameters must agree in:

- Number
- Type (relative to legal coercions)

Parameter Passing

A formal parameter is allocated space in the method and the value of the actual parameter is copied into that space.

- Primitive data types: Pass by value.
- Objects:

  A reference to the object is passed by value.

  Formal parameter has a reference (pointer) to the actual object.

Question: Does Java implement true pass by reference for objects?

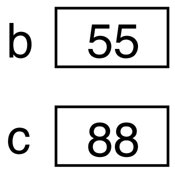
Example: Swap

Swap in Java

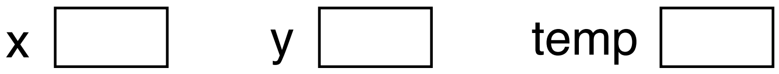
```
static void swap (int x, int y)
{
        int temp = x;
        x = y;
        y = temp;
}
```

Call:

```
int b = 55;
int c = 88;
swap(b,c);
```



In swap:



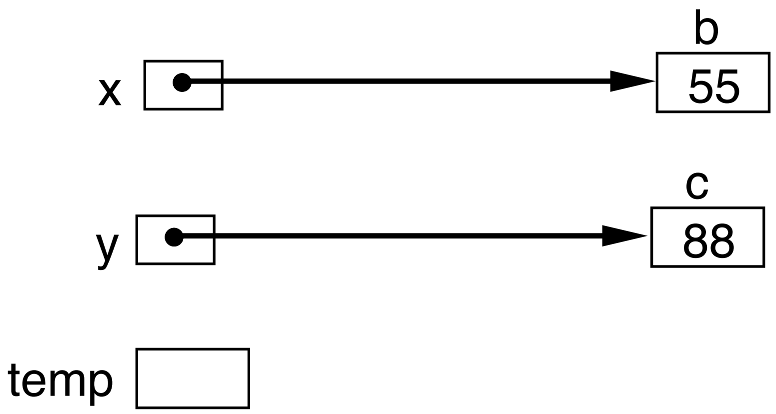
No change in b or c!

Swap in C++

```
void swap (int &x, int &y)
{
	int temp = x;
	x = y;
	y = temp;
}
```

Call:
```
int b = 55;
int c = 88;
swap(b,c);
```

In swap:



The swap occurs.

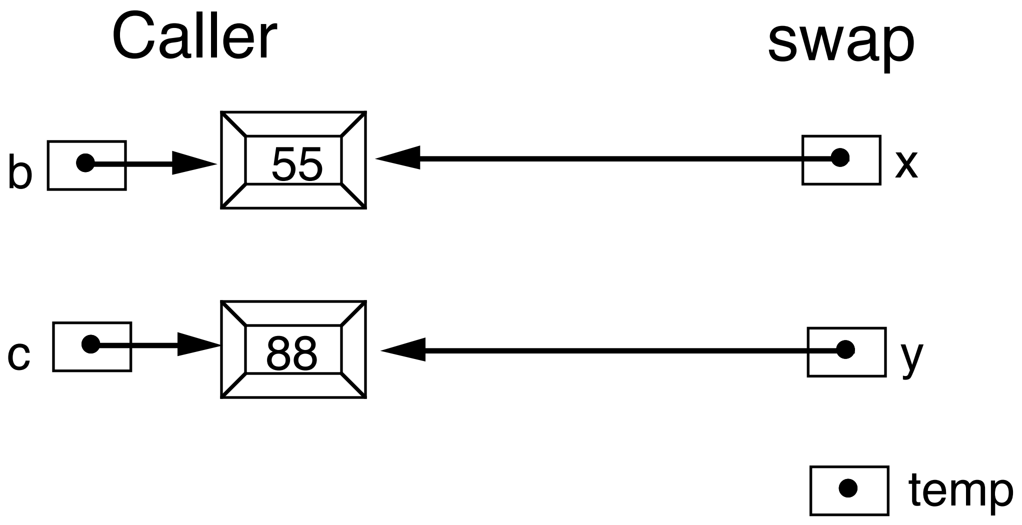
Call this effect **alias reference**.

But Java does not allow us to pass primitive variables by reference because the aliasing may compromise security.

Try “pass by reference” with Objects.

```
static void swap (Integer x, Integer y)
{
	Integer temp = x;
	x = y;
	y = temp;
}
```

```
Call:   Integer b = new Integer(55);
        Integer c = new Integer(88);
        swap(b,c);
```



No swap!

Call this effect **copy reference**.

Note: The value inside an Integer object is hidden.

Build our own Integer Objects.

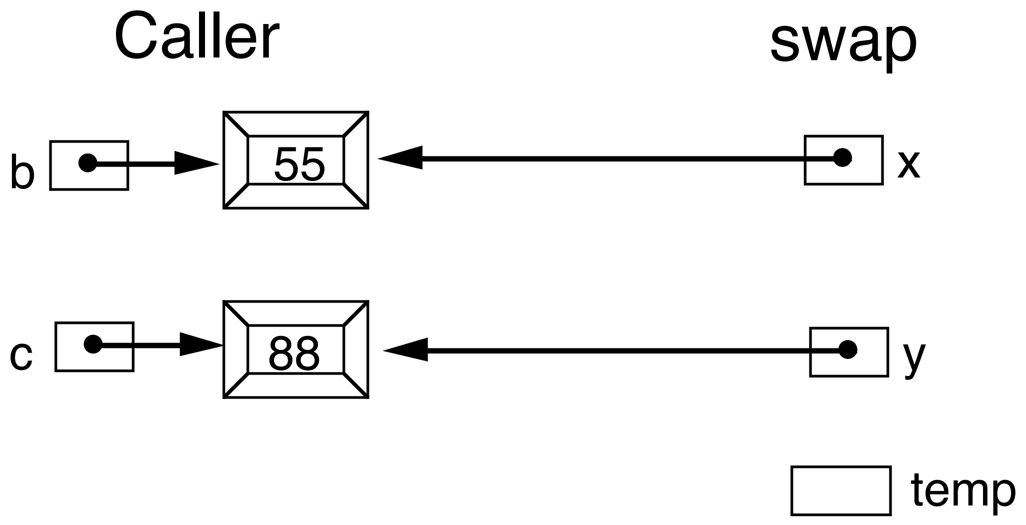
```
public class Int
{
        public int myInt;                       // instance variable

        public Int (int m)                      // constructor
        {  myInt = m;  }

        public int getValue()                   // selector or
        {  return myInt;  }                     // accessor

        public int setValue (int m)             // mutator
        {  myInt = m;  }
}
```

Try swap again:

```
static void swap (Int x, Int y)
{
        int temp = x.myInt;
        x.myInt = y.myInt;
        y.myInt = temp;
}
```

Call:

```
Int b = new Int(55);
Int c = new Int(88);
swap(b,c);
```



Swap occurs because we have access inside the objects.

This effect can be obtained even if the instance variable is private *if* we have accessor and mutator methods.

```
static void swap (Int x, Int y)
{
        int temp = x.getInt();
        x.setInt(y.getInt);
        y.setInt(temp);
}
```

Array Parameters

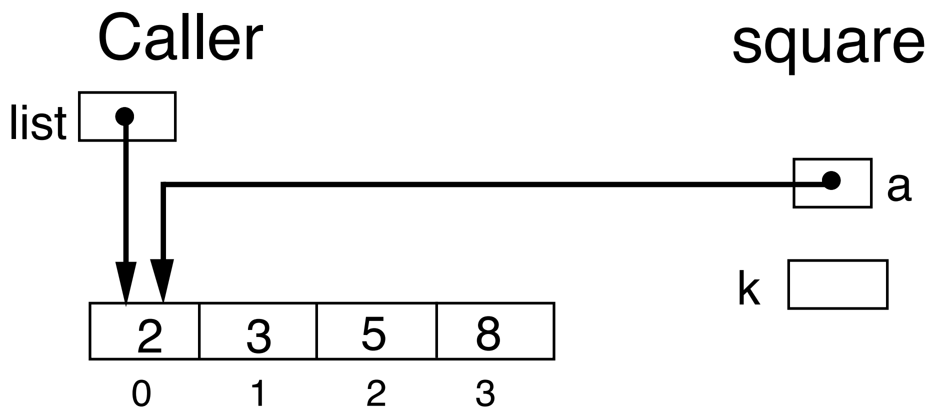
The components of an array act as public instance variables.

Example

A method that takes an array of **int** and squares each component.

```
static void square (int [] a)
{
        for (int k = 0; k < a.length; k++)

                    a[k] = a[k] * a[k];
}
```

Call: **int** [] list = { 2, 3, 5, 8 };
square (list);



This property means we can sort arrays.

Return Value for a Method

void means method is a *procedure*.

- Its execution is for side effect only.
- Output or changes in nonlocal data.

Any actual type means method is a *function*.

- Method must contain command(s)

  ```
  return expr;
  ```

  where expr is an expression of an type compatible with the declared return type.

- Functions are normally used in expressions where their value is consumed.

  ```
  m = Math.random();
  k = m + 2*Integer.parseInt("123");
  ```

- Possible return types

  Primitive data types

  Array types

  Any class

- Functions should *not* have side-effects normally.

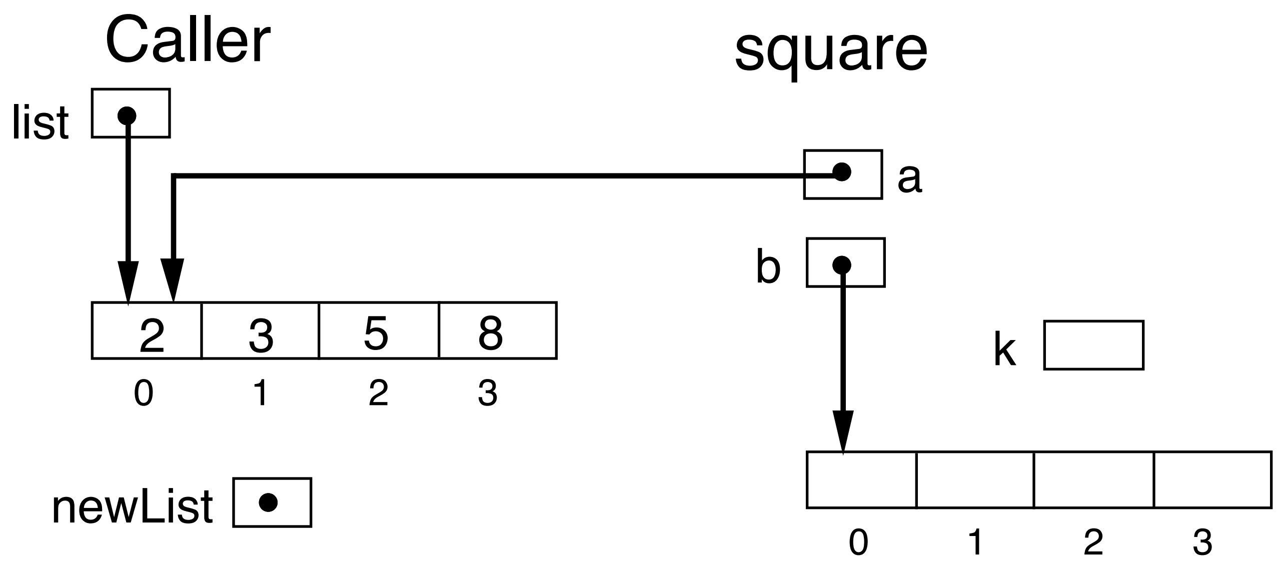
Example

A method that takes an array of **int** and returns an array containing their squares.

```
static int [] square (int [] a)
{
        int [] b = new int [a.length];
        for (int k = 0; k < a.length; k++)
                        b[k] = a[k] * a[k];
        return b;
}
```

Call:

```
int [] list = { 2, 3, 5, 8 };
int [] newList = square(list);
```



Note that original array is unchanged.

Functional programming works like this.

Strings

Strings in Java are provided by the class String, whose objects are immutable sequences of characters: Once created a string cannot be altered.

(Use the class StringBuffer for mutable strings.)

public final class java.lang.String **extends** java.lang.Object

Some String Constructors

```
    public String ();
    public String (String value);
    public String (char [] value);
```

Examples

```
String s1 = new String();          // empty string
char [] chs = { 'h', 'e', 'r', 'k', 'y' };
String s2 = new String(chs); // "herky"
String s3 = new String(s2);        // a copy of s2
String s4 = s2;                              // same reference
String s5 = "Gateway to Nebraska";
```

Note that this string is created without using **new**.

"Gateway to Nebraska" is an anonymous String object.

Length of a String

An instance method

```
            public int length();
        s1.length()        returns 0
        s2.length()        returns 5
```

Comparing Strings

```
    public boolean equals (Object anObject);
```

public boolean equalsIgnoreCase (String other);
public int compareTo (String other);

s2 == s3	returns false
s2.equals(s3)	returns true
s2 == s4	returns true
s2.equalsIgnoreCase("Herky")	returns true
s2.compareTo(s3)	returns 0
s3.compareTo(s5)	returns 33

Note: Value of 'h' = 104
Value of 'G' = 71
s3.compareTo(s5) > 0 when s3 $>_{lex}$ s5
s3.compareTo(s5) < 0 when s3 $<_{lex}$ s5

Characters in Strings

Strings in Java are not arrays, so they cannot be subscripted using brackets [ ].

The method

public char charAt (**int** index);

provides the character the position index (zero based).

```
for (int k=0; k<s2.length(); k++)
        System.out.println(s2.charAt(k\));
```

Concatenation: Infix +

Observe the consequences of precedence and overloading.

```
System.out.println(" " + 5 + 8);      // prints 58
System.out.println(5 + " " + 8);      // prints 58
System.out.println(5 + 8 + " ");      // prints 13
```

Trim

Instance method

public String trim();

removes leading and trailing whitespace.

"   123.45   ".trim() returns "123.45"

Other String Methods

In the context of concatenation, values of the primitive types are coerced to String using the method valueOf.

> **public static** String valueOf (**int** i);
>
> **public static** String valueOf (**long** l);
>
> **public static** String valueOf (**float** f);
>
> **public static** String valueOf (**double** d);
>
> **public static** String valueOf (**boolean** b);
>
> **public static** String valueOf (**char** c);

In the command,

```
System.out.println("k = " + k);
```

the **int** k is coerced to String using valueOf before the concatenation.

The method toString performs the corresponding coercion for objects.

In addition, String has methods for finding characters and substrings in a string and for copying a string into an array.

> **int** indexOf(String str)
>
> **int** indexOf(String str, **int** fromIndex)
>
> String substring(**int** start)
>
> String substring(**int** start, **int** pastEnd)
>
> **char** charAt(**int** index)
>
> **void** getChars(**int** start, **int** pastEnd, **char** [] dst, **int** dstBegin)
>
> String concat(String otherStr)
>
> String toLowerCase()
>
> String toUpperCase()
>
> **int** compareTo(String otherStr)
>
> **boolean** equals(Object anObject)

Methods in Character

> **public static boolean** isWhitespace(**char** ch);
> **public static boolean** isDigit(**char** ch);
> **public static boolean** isLetter(**char** ch);
> **public static boolean** isLetterOrDigit(**char** ch);
> **public static boolean** isLowerCase(**char** ch);
> **public static boolean** isUpperCase(**char** ch);