The Java Language

From Sun
- Reworking of C++
  - Reworking of C

Cleans up C++
- Adds pointer safety
- Strong, static type checking
- Garbage collection
- Exception handling
- Compiles to bytecodes
  - Virtual machine / interpreted
  - Platform independence
    - WWW use

Unicode

Character Set

16-bits per character

Mostly transparent

```
x = “ABC\n>>>>\u04ef<<<<”;
```

Strings may NOT include the newline directly.

Comments

```
// This is a comment
/* This is a comment */
/** This is a comment */
```

Comments do not nest

```
/* Ignore this code...
i = 3;
j = 4;    /* This is a comment */
k = 5;
*/
```

Primitive Data Types

boolean	
char	*16-bit Unicode character*
byte	*8-bit integer*
short	*16-bit integer*
int	*32-bit integer*
long	*64-bit integer*
float	*32-bit floating point*
double	*64-bit floating point*

Similar to “C”s basic types

boolean is not an integer

```
        int i = 1;
        if (i) …        // Illegal
```

char is 16 bits, not 8

byte is 8 bit integer

Boolean

Two constants (literals):

```
    true
    false
```

Cannot convert between integer and boolean

Cannot even cast.

```
        boolean b = …;
        int i;
        i = (int) b;      // error
        b = (boolean) i;  // error
```

Operators (just as in “C”):

`!`	Logical negation
`==   !=`	Equals, not-equals
`&   |   ^`	Logical “and,” “or,” and “x-or” (both operands evaluated)
`&&   ||`	Logical “and” and “or” (short-circuit evaluation)
`?:`	Ternary conditional operator
`=`	Assignment
`&=   |=   ^=`	The operation, followed by assignment

Numbers - Similar to “C”

Literals

`123`	Decimal
`0x7B`	Hex
`0173`	Octal
`123L`	Long

```
12.34f
12.34F
12.34d
12.34D
```

Data types:

byte	8-bits
short	16-bits
int	32-bits
long	64-bits
float	32-bits
double	64-bits

Operators (all)

Same precedence as “C”, “C++”

highest	`[]` `.` `(`*params*`)` *expr*`++` *expr*—
	`++`*expr* `--`*expr* `+`*expr* `-`*expr* `~` `!`
	`new` `(`*type*`)`*expr*
	`*` `/` `%`
	`+` `-`
	`<<` `>>` `>>>`
	`<` `>` `<=` `>=` `instanceof`
	`==` `!=`
	`&`
	`^`
	`|`
	`&&`
	`||`
	`?:`
lowest	`=` `+=` `-=` `*=` `/=` `%=` `<<=` `>>=` `>>>=` `&=` `^=` `|=`

New Operators

```
x = new Person (…)
x instanceof Person
x . foo (a, b, c)
x . field
```

Messages sends are left-associative:

What does

```
          x .  f ()  .  g ()  .  h ()
```

mean?

```
       (((x .  f ()) .  g ()) .  h ()
```

Field accessing is left-associative:

```
       w  .  a  .  b  .  c
  = ((w  .  a) .  b) .  c
```

Casting Rules

Implicit conversions: Inserted by the compiler

```
char → short
byte → short
short → int
int → long
long → float
float → double
```

Example:

```
int i = …;
float f = …;
f = i;          // conversion inserted by compiler
```

Explicit casting

Example:

```
i = (int) f;    // must use a cast here
```

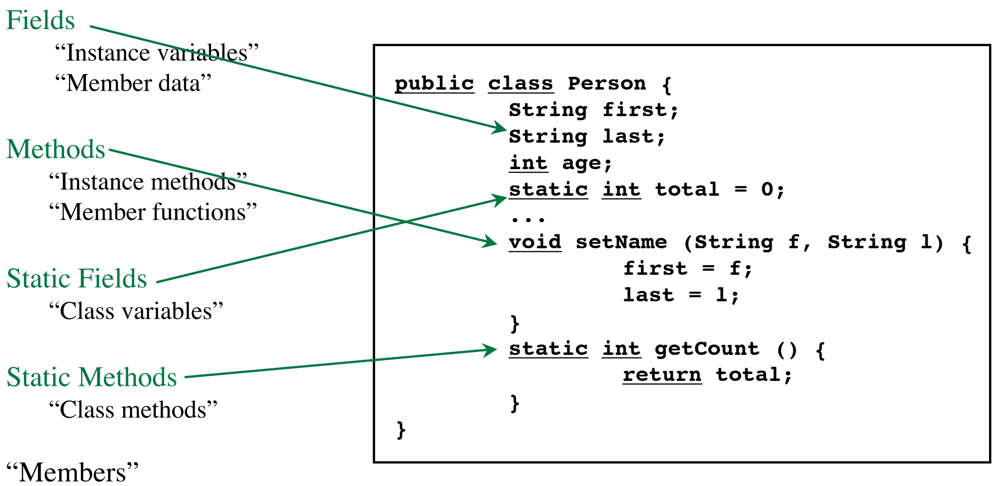
Some things may not be cast

```
int i = …;
Person p = …;
p = (Person) i;         // illegal
i = (int) p;            // illegal
```

An Example Class

```
public class Person {
    String first;
    String last;
    int age;
    static int total = 0;
    Person (String f, String l, int a) {
        first = f;
        last = l;
        age = a;
        total++;
    }
    String getName () {
        return last + ", " + first;
    }
    void setName (String f, String l) {
        first = f;
        last = l;
    }
    static int getCount () {
        return total;
    }
}
```

Terminology



Fields
    “Instance variables”
    “Member data”

Methods
    “Instance methods”
    “Member functions”

Static Fields
    “Class variables”

Static Methods
    “Class methods”

```
public class Person {
    String first;
    String last;
    int age;
    static int total = 0;
    ...
    void setName (String f, String l) {
        first = f;
        last = l;
    }
    static int getCount () {
        return total;
    }
}
```

“Members”

Fields and methods

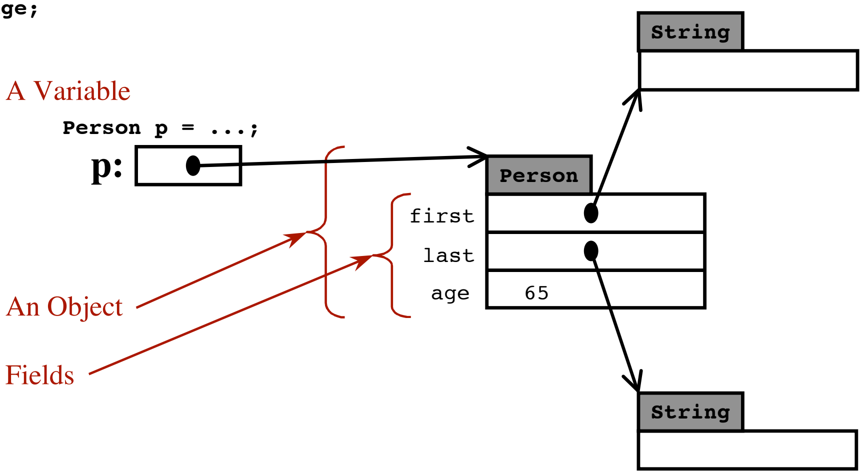
Static fields and static methods

Variables and Objects

```
public class Person {
    String first;
    String last;
    int age;
    ...
}
```

A Variable

```
Person p = ...;
```



Pointers to Objects

Classes are related in a subclass - superclass hierarchy.

```
    class Person { … }
    class Student extends Person { … }

    Person p = ... ;
    Student s = ... ;
```

Variable p will always point to an instance of Person or one of its subclasses.

(Or it might be “null”)

Each assignment is checked.

“Assignment compatibility”

```
    p = s;         // OK
    s = p;         // Error: compiler will flag this
```

Must use a cast:

```
    s = (Student) p;
```

Dereferencing Pointers

In C and C++:

```
struct MyType { ... };
MyTpye *p, *q;
```

In Java:

```
class MyClass { ... };
MyClass p, q;
```

Dereferencing Pointers

In C and C++:

```
struct MyType { ... };
MyTpye *p, *q;

(*p).field = (*q).field;   /* Get from memory & store into memory */
```

In Java:

```
class MyClass { ... };
MyClass p, q;

p.field = q.field;         /* Get from memory & store into memory */
```

Dereferencing Pointers

In C and C++:

```
struct MyType { ... };
MyTpye *p, *q;

(*p).field = (*q).field;    /* Get from memory & store into memory */

p = q;                      /* Copy the pointer */
```

In Java:

```
class MyClass { ... };
MyClass p, q;

p.field = q.field;          /* Get from memory & store into memory */

p = q;                      /* Copy the pointer */
```

Dereferencing Pointers

In C and C++:

```
struct MyType { ... };
MyTpye *p, *q;

(*p).field = (*q).field;    /* Get from memory & store into memory */

p = q;                      /* Copy the pointer */
*p = *q;                    /* Copy the structs */
```

In Java:

```
class MyClass { ... };
MyClass p, q;

p.field = q.field;          /* Get from memory & store into memory */

p = q;                      /* Copy the pointer */
p.copyFieldsFrom(q);        /* To copy data, you must write code */
```

Dereferencing Pointers

In C and C++:

```
struct MyType { ... };
MyTpye *p, *q;

(*p).field = (*q).field;    /* Get from memory & store into memory */

p = q;                      /* Copy the pointer */
*p = *q;                    /* Copy the structs */

if (p == q) ...             /* Compare pointers */
```

In Java:

```
class MyClass { ... };
MyClass p, q;

p.field = q.field;          /* Get from memory & store into memory */

p = q;                      /* Copy the pointer */
p.copyFieldsFrom(q);        /* To copy data, you must write code */

if (p == q) ...             /* Compare pointers */
```

Dereferencing Pointers

In C and C++:

```
struct MyType { ... };
MyTpye *p, *q;

(*p).field = (*q).field;    /* Get from memory & store into memory */

p = q;                      /* Copy the pointer */
*p = *q;                    /* Copy the structs */

if (p == q) ...             /* Compare pointers */
if (*p == *q) ...           /* Compare two structs */
```

In Java:

```
class MyClass { ... };
MyClass p, q;

p.field = q.field;          /* Get from memory & store into memory */

p = q;                      /* Copy the pointer */
p.copyFieldsFrom(q);        /* To copy data, you must write code */

if (p == q) ...             /* Compare pointers */
if (p.equals(q)) ...        /* Compare two objects */
```

Equality Testing

Assignment

```
        p = s
```

Testing

Compares pointers, does not chase the pointers to the data

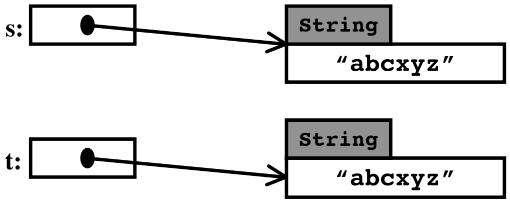
```
        p == s
        p != s
```

Examples (same as in "C"):

```
        if (p == s) { ... }        // compares pointers
        if (p = s) { ... }         // assignment is an expression
        p = s;                     // expression, used as a statement
        p == s;                    // also legal
```

More examples:

```
    String s, t = ...;
    s = "abc" + "xyz";
    if (s == t) ...;
    if (s.equals (t)) ...;
    if (s == "abcxyz") ...;
    if (s.equals ("abcxyz")) ...;
```



instanceof

The "instanceof" operator can test the class of an object:

```
    x = new Student (...);
    if (x instanceof Student) ...      // true
    if (x instanceof Person) ...       // also true
```

The compiler treats:

```
    s = (Student) p;
```

Like this:

```
    if (p instanceof Student) {
          s = p;
    } else {
          throw new ClassCastException ();
    }
```

You could always code this explicitly…

```
    if (p instanceof Student) {
          s = (Student) p;
    } else {
          ... Do something else ...
    }
```

Garbage Collection

Built-in garbage collector
Every word contains either:
    Pointer
    Primitive data value
All pointers in object memory can be identified by GC.
Objects can be moved.
All pointers can be readjusted
    … in the middle of program execution.
The Java programmer can never know where things are in memory.
Example from C:

```
int * addr;
addr = (int *) 0x1234abcd;
x = *addr;
*addr = x;
```

Some C programmers use this ability:

```
Person p [] = ...;
i = (int)  & (p[5]);
i = i + 17 * (sizeof Person);
((Person *) i) -> field = ...;
```

What if GC happens here?

Difficult to garbage collect in C++.

Statement Syntax

Just like C and C++

Assignment Statement:

```
x = y + 5;
```

Expressions can be used as statements.

Increment and decrement statements:

```
i++;
```

Message sending (method invocation):

```
p.foo (a, b, c);
(y.meth()).foo (x.bar(), b+4, c.meth2());
```

(If method is not void, the returned value is ignored.)

Object creation

```
new Person ("Harry", "Porter");
```

A reference to this object was not saved; more likely:

```
p = new Person ("Harry", "Porter");
```

Flow of Control Statements - If

The if statement:

```
if (boolean-expression)
        statement-1;
else
        statement-2;
```

Optional

Statement blocks:

```
{
        statement;
        statement;
        statement;
}
```

Example:

```
if (boolean-expression) {
        statement;
        statement;
        statement;
} else {
        statement;
        statement;
        statement;
}
```

Flow of Control Statements - Looping

```
while (boolean-condition) {
   statement;
   statement;
   statement;
}


for (i=0,j=100; i<5; i++,j--) {
   statement;
   statement;
   statement;
}


do {
   statement;
   statement;
   statement;
} while (boolean-condition);
```

Flow of Control Statements - Switch

```
switch (integer-expression) {
    case 23:
            statement;
            statement;
            break;
    case 45:
            statement;
            statement;
            break;
    case 51:
    case 52:
    case 53:
            statement;
            statement;
            break;
    default:
            statement;
            statement;
            break;
}
```

Flow of Control Statements - Misc

Break, continue, and labels:

```
                  while (condition-1) {
my_loop:              while (condition-2) {
                          while (condition-3) {
                              ...
                              break my_loop;
                              ...
                              continue my_loop;
                              ...
                          }
                      }
                  }
```

The return statement:

```
return;
return expression;
```

Arrays

Examples:

```
Person [] p;
Person [] p = new Person [10];
... x.foo (i, new Person [10], j) ...   // In any expression
```

Older C syntax for array declarations:

```
Person p [] = new Person [10];
```

Numbering starts at 0:

```
p[0], p[1], ... , p[9]
```

Initialization Examples:

```
Person [] p = new Person [10];
for (int i =0; i<p.length; i++) {
     p[i] = new Person(...);
}

int [] [] a = { {1, 2}, {4, 5, 6}, {3}};

int [] [] a = {
                {1, 1, 4, 1, 1, 1},
                {1, 1, 5, 1, 1, 1},
                {1, 1, 6, 1, 1, 1},
              };
```

Strings

A predefined class: String

```
String x = "hello";
System.out.print (x);
System.out.println (x);
```

String Concatenation

```
x = x + " there";
System.out.println ("The value is " + i);
System.out.println ("The value is " + (i.toString()));
```

Predefined functions:

```
x.length () ➔ int
x.charAt (int) ➔ char
x.indexOf (char) ➔ int
x.equals (String) ➔ boolean
x.equalsIgnoreCase (String) ➔ boolean
x.startsWith (prefixString) ➔ boolean
x.endsWith (suffixString) ➔ boolean
x.compareTo (String) ➔ -1,0,+1
x.substring (startPos,endPos) ➔ String
x.toLowerCase () ➔ String
x.toUpperCase () ➔ String
x + y ➔ String
x.toCharArray () ➔ char[]
```

String and StringBuffer

Strings are immutable.

StringBuffer
    Like String with mutation

Constructors:

```
StringBuffer (String) ➔ StringBuffer
StringBuffer (initCapacity) ➔ StringBuffer
StringBuffer () ➔ StringBuffer
```

Methods:

```
StringBuffer x =  ...;
x.append (y) ➔ StringBuffer
x.setCharAt (int,char) ➔ void
x.setLength (int) ➔ void
```

Classes

Class names are capitalized.

Modifiers of a class

```
        public
        abstract
        final
        strictfp

public class MyClass1 { … }
abstract class MyClass2 { … }
final class MyClass3 { … }
public abstract class MyClass4 { … }
```

Fields and methods may have modifiers

```
        public
        private
        protected
        static
        volatile

static int getCount () { … }
public void foo (…) { … }
static final double pi = 3.1415;
```

Constructors

```
public class Person {
      ...
      Person (String f, String l, int a) {
                first = f;
                last = l;
                age = a;
                total++;
      }
      Person () {
                first = "John";
                last = "Doe";
                age = 0;
                total ++;
      }
      Person (String f, String l) {
                this (f, l, 0);
      }
      ...
}


Person p;
p = new Person ("Harry", "Porter", 50);

Person p = new Person ("Harry", "Porter", 50);
Person q = new Person ();
Person r = new Person (“Harry”, “Porter”);
```

Constructors

The sequence of events:

- The object is created.
- All fields are initialized to default values.
  int --> 0
  float --> 0.0
  object references --> null
  boolean --> false
  char --> ‘\u0000’
- Initializing expressions are executed.
- Constructor is invoked.

```
class MyClass {
    String name;
    String addr;
    int age = 123;
    int ssNum = ssGen.getNext ();
    …
    MyClass (String n, String a) {
        name = n;
        addr = a;
    }
    MyClass (String n) {
        this (n, “<no address>”);
        ... other stuff ...
    }
}
```

Constructor may invoke other constructors.

The “no arg” constructor.

Insufficient Memory?
VM will throw “OutOfMemoryError”

Null Pointers

Pre-defined indentifiers: “null”, “true”, “false”
(Not keywords)

Null is a value.
Imagine storing 0 in the variable.

```
Person p;
…
p = null;
…
t = p . computeTax (2004);
p.name = “Fred”;
```

What happens?
The “NullPointerException” is thrown.
VM must test every use.

Arrays are objects, too.
(array variables can be null)

```
int [] a;
…
a [i] = a [j];
…
a = { 1, 2, 3 };
```

Alternative (e.g., ST):
Pointer to a special object.
Can deal with ALL messages (by invoking error handling)

“this”

```
class Person {
  ...
  void foo () {
    ...
    this.bar ();

    ... this.name ...

  }
  ...
}
```

“this”

```
class Person {
  ...
  void foo () {
    ...
    this.bar ();
    bar ();             // equivalent
    ... this.name ...
    ... name ...        // equivalent
  }
  ...
}
```

“this” and “super”

Both refer to the receiver

```
class Person {
  ...
  void foo () {
    ...
    this.bar ();
    bar ();             // equivalent
    ... this.name ...
    ... name ...        // equivalent
  }
  ...
}

class Student extends Person {
  ...
  void foo () {          // overrides the inherited version
    ...
    foo ();              // invoke this method, recursively
    ...
    super.foo ();        // invoke the overridden version
    ...
    super.bar ();
    ...
  }
  ...
}
```

Classes May Implement Interfaces

```
interface TaxableEntity {
  String getName ();
  int getID ();
  int computeTax (int year);
}


class Person implements TaxableEntity {
  ...
  String getName () { ... }
  int getID () { ... }
  int computeTax (int year) { ... }
  ...
}


class Corporation implements TaxableEntity {
  ...
  String getName () { ... }
  int getID () { ... }
  int computeTax (int year) { ... }
  ...
}
```

Interfaces

Example:

```
interface MyInter extends OtherInterA, OtherInterB, OtherInterC {
     int foo (...);
     int bar (...);
     void myFunct (...);
     ...
     int x = 123;
     double pi = 3.1415;
}
```

Message:

```
void myFunct (int a, char ch);
```

Method:

```
void myFunct (int a, char ch) { ... statements ... }
```

Interfaces can contain:

- Messages
- Constants

Interfaces

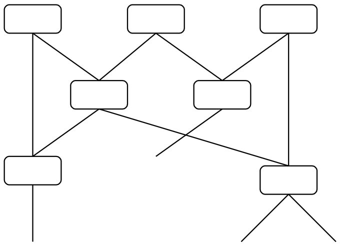
Each interface extends zero or more interfaces.

Example:

```
interface MyInter extends InterA, InterB, InterC {
      ...
}
```

Example:

```
interface NoParentInterf { ... }
```



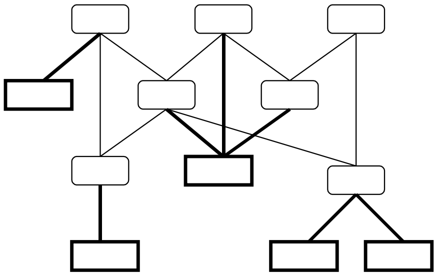
Interfaces are organized in a hierarchy.
- Sub-interface / super-interface
- Directed, Acyclic Graph (DAG)
  - Multiple, not single "inheritance"
- No single root

Classes Implement Interfaces

Each class implements zero or more interfaces.

Example:

```
class MyClass implements InterA, InterB, InterC {
      ...
}
```

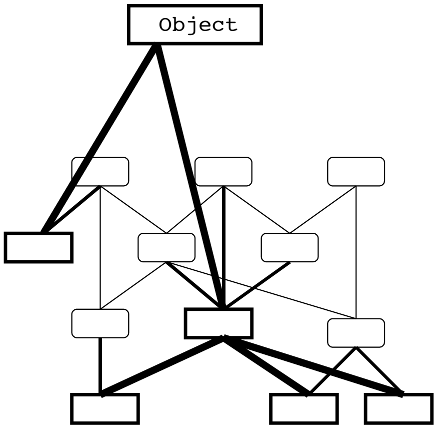


Classes Extend Classes

Each class extends exactly one other class.

```
class MyClass extends MySuper
    implements InterA, InterB,... {
    ...
}
```

The subclass hierarchy.
Tree-shaped.
Class “Object” is the root class.



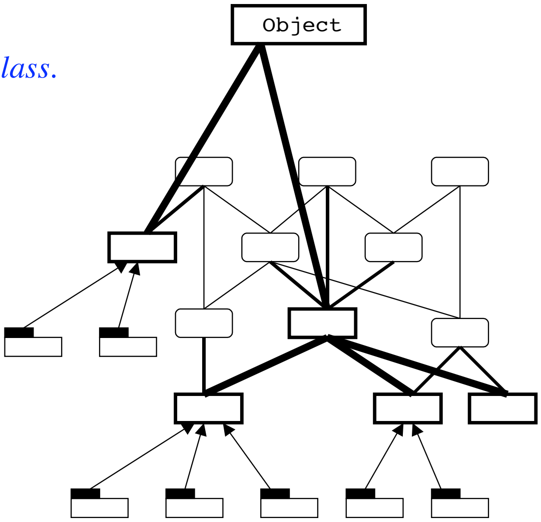
© Harry H. Porter, 2005

Objects are Instances of Classes

Each object is an instance of one class.

Types of Relationships:

Instance-of
Extends (subclass)
Extends (subinterface)
Implements



© Harry H. Porter, 2005

Types

Types:

Primitive types (int, double, char, boolean, …)
Classes (e.g., Person, Student)
Interfaces (e.g., TaxableEntity)

Basic Syntax of Declarations:

<modifiers> <type> <id> [= <expr>] , … , <id> [= <expr>] ;

```
int i, j, k;
int i=1, j=2, k=3;
static final double pi = 3.1415;

TaxableEntity t;
t = new Person (...);

Person p;
t = p;
p = t;   // Error
```

Modifiers on Fields & Variables

public
private
protected
<package> (No keyword, the default)

Access Control

static
Fields: A class variable, not an instance variable
Variables in Methods: One copy, value retained across invocations

final
A constant value.
Value will be assigned only once.
Can be used on instance fields, class variables, local variables, parameters

volatile
For multi-threaded programs.
Variable may be shared
The compiler should generate code to fetch and store variable immediately
Must not cache this variable in registers

Access Control: Public

No control:
- Fields may be accessed from any code.
- Methods may be invoked from any code.

```
class MyClass {
   public String name;
   public void foo (…) { … }
   …
}

class AnotherUnrelatedClass {
   …
   void method () {
        MyClass x;
        …
        x.name = “Santa Claus”;
        x.foo (…);
        …
   }
}
```

Access Control: Private

The most restrictive:
- Fields: can only be accessed from code in this class.
- Methods: can only be invoked from code in this class.
- Even code in subclasses is prohibited from using private stuff.

```
class MyClass {
   private String name;
   private void foo (…) { … }
   …
   void method () {
        MySub x;
        …
        x.name = “Santa Claus”;
        x.foo (…);
        …
        name = “Joe”;
        foo (…);
        …
   }
}
```

Okay

```
class MySub extends MyClass {
    …
    void method () {
       MyClass x;
       …
       x.name = “Santa Claus”;
       x.foo (…);
       …
       name = “Joe”;
       foo (…);
       …
    }
}
```

Errors!

Access Control: Package

Every class belongs to exactly one “package.”
    A package contains several classes and interfaces.
    The unit of program development.

“Package” access is the default:
    Fields: can only be accessed from code in this package.
    Methods: can only be invoked from code in this package.

```
class MyClass {
    String name;
    void foo (…) { … }
    …
    void method () {
          MySub x;
          …
          x.name = “Santa Claus”;
          x.foo (…);
          …
          name = “Joe”
          foo (…);
          …
    }
}
```

```
class MySub extends MyClass {
    …
    void method () {
       MyClass x;
       …
       x.name = “Santa Claus”;
       x.foo (…);
       …
       name = “Joe”;
       foo (…);
       …
    }
}
```

Depends on which package this class is in.

Access Control: Protected

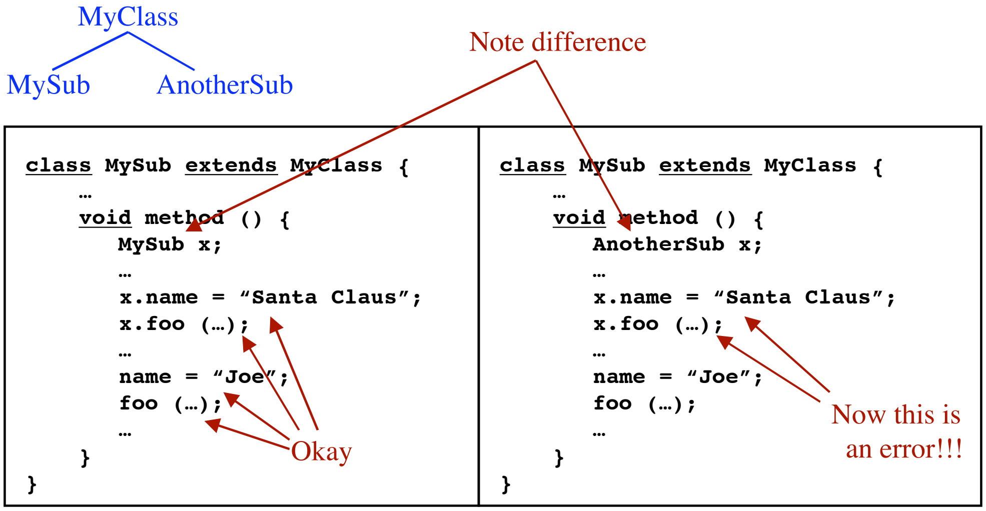
May be accessed by code in this class and its subclasses.

```
class MyClass {
    protected String name;
    protected void foo (…) { … }
    …
    void method () {
          MySub x;
          …
          x.name = “Santa Claus”;
          x.foo (…);
          …
          name = “Joe”
          foo (…);
          …
    }
}
```

```
class MySub extends MyClass {
    …
    void method () {
       MySub x;
       …
       x.name = “Santa Claus”;
       x.foo (…);
       …
       name = “Joe”;
       foo (…);
       …
    }
}
```

Okay

Access Control: Protected



MyClass

MySub AnotherSub

Note difference

```
class MySub extends MyClass {
    …
    void method () {
       MySub x;
       …
       x.name = “Santa Claus”;
       x.foo (…);
       …
       name = “Joe”;
       foo (…);
       …
    }
}
```

Okay

```
class MySub extends MyClass {
    …
    void method () {
       AnotherSub x;
       …
       x.name = “Santa Claus”;
       x.foo (…);
       …
       name = “Joe”;
       foo (…);
       …
    }
}
```

Now this is an error!!!

© Harry H. Porter, 2005

Running a Java Program

Filename “Echo.java”:

```
class Echo {
  public static void main (String[] args) {
    System.out.println("Welcome!");
    for (int i = 0; i<args.length; i++) {
      System.out.print (args[i] + " ");
    }
    System.out.println();
  }
}
```

To run this program in Unix:

```
% addpkg
% javac Echo.java
% java Echo Hello there
Welcome!
Hello there
%
```

© Harry H. Porter, 2005

Packages

A set of related classes and interfaces

Package = Unit of Program Development

Each package is named

Dot notation: `com.sun.java.games`

First line in a file should be:

```
package x.y.z;
```

If missing? “unnamed” package

Important Packages:

java.lang	Essential classes; always imported automatically
java.io	Basic I/O (files and character streams)
java.util	Data structure classes
java.awt	“Abstract Windowing Toolkit” (user interface classes)
java.net	Sockets, TCP/IP, and URL classes
java.applet	Running Java programs over WWW and internet browsers

Using Packages

Package “java.util” contains “Date Class”

Can use it in (code in) another package.

Must give “fully qualified” names:

```
java.util.Date d = new java.util.Date ();
```

Importing stuff from a package:

```
import java.util.Date;
```

Now we can write:

```
Date d = new Date ();
```

To import everything from a package:

```
import java.util.*;     // to import everything in the package
```

Exceptions

Runtime errors occur!!!

```
x = a[i+1];
```

Names of predefined exceptions:

```
ArrayIndexOutOfBoundsException
NullPointerException
ClassCastException
… etc …
```

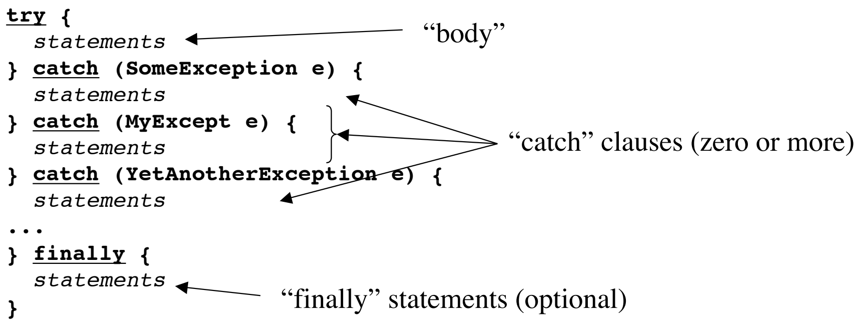
Each exception is modeled with a class.
Can add new exception classes:

```
class MyExcept extends Exception {
 ...
}
```

Can "throw" an exception:

```
throw new MyExcept ();
```

The "try" Statement



```
try {
  statements
} catch (SomeException e) {
  statements
} catch (MyExcept e) {
  statements
} catch (YetAnotherException e) {
  statements
...
} finally {
  statements
}
```

"body"

"catch" clauses (zero or more)

"finally" statements (optional)

Passing Data to the Catch Clause

Use fields in the exception class
Provide a constructor that takes arguments

```
class MyExcept extends Exception {
  String severity;
  MyExcept (String s) { severity = s; }
  ...
}
```

Provide arguments to the constructor:

```
try {
  ...
  throw new MyExcept (“Mission-Critical”);
  ...
} catch (MyExcept e) {
  ... use e.severity here ...
}
```

Try statements may be nested

```
try {
  ...
    try {
      ...
      throw ...;
      ...
    } catch (...) {
      statements
    } catch (...) {
      statements
    ...
    }
  ...
} catch (...) {
  statements
} catch (...) {
  statements
...
}
```

The error will propagate upward / outward
… until caught.

Exceptions propagate through methods

```
bar () {
  ...
    try {
      ...
      x.foo ();
      ...
    } catch (...) {
      statements
    } catch (...) {
      statements
    ...
    }
  ...
}
```

```
foo () {
  ...
    try {
      ...
      throw ...;
      ...
    } catch (...) {
      statements
    } catch (...) {
      statements
    ...
    }
  ...
}
```

Catch Clauses

May finish by:

- Throwing a new exception
  Other catch clauses in this “try” are NOT used.
- Execute a return statement
- Normal completion
  Execution “falls through” to code after the “try” statement

The “finally” Statements

Will always be executed
  after “try” statements
  after “catch” clause (if one was executed)

Doesn’t matter how the “body” statements ended...

- Normal (fall through)
- Exception thrown
- Return statement

“Finally” statements may…

- Throw a new execption
  (it overrides previous exception or return, if any)
- Execute a return statement
  (it overrides previous exception or return, if any)
- Normal (fall through)
  (continue with exception, return, or normal exit)

Contracts and Exceptions

Each method includes:

- A name (i.e., a selector)
- Number and types of arguments
- Return value
- Exceptions that they might throw

```
public void foo (...)
       throws MyExcept, AnotherExcept, YetAnotherException
  {
    ...
    throw new myExcept (...);
    ...
  }
```

If a method’s body MAY throw exception E…

The method must either

- Catch E
- List E in method header

```
public void bar (...)
       throws YetAnotherException
  {
    ...
    try {
      ... foo (...) ...
    } catch (MyExcept e) {
      ...
    } catch (AnotherExcept e) {
      ...
    }
  }
```

Implementing OOP Languages

Object = Block of memory (i.e., “struct”, “record”)
Field = Offset into record

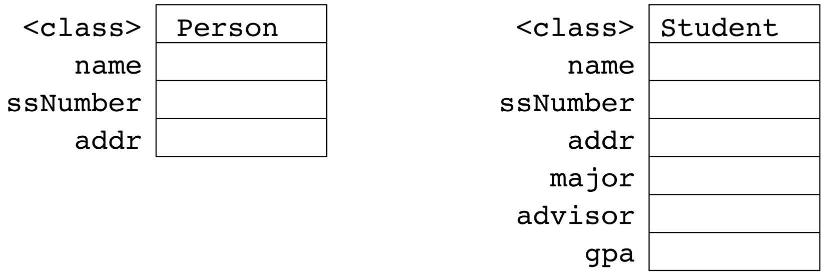
The first (hidden) field indicates the class of the object.

<class>	Person
name	
ssNumber	
addr	

Implementing OOP Languages

Subclassing:
Existing fields in the same locations
New fields added to end of record

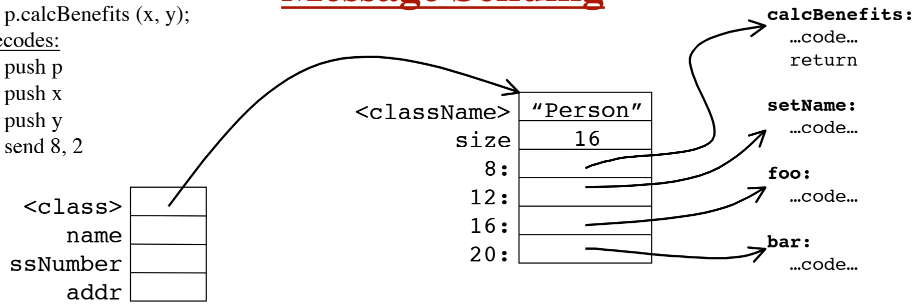
Example: Student is a subclass of Person


<class> Person
name
ssNumber
addr

<class> Student
name
ssNumber
addr
major
advisor
gpa

Message Sending

Source:
p.calcBenefits (x, y);
Bytecodes:
push p
push x
push y
send 8, 2


<class>
name
ssNumber
addr

<className> "Person"
size 16
8:
12:
16:
20:

calcBenefits:
…code…
return

setName:
…code…

foo:
…code…

bar:
…code…

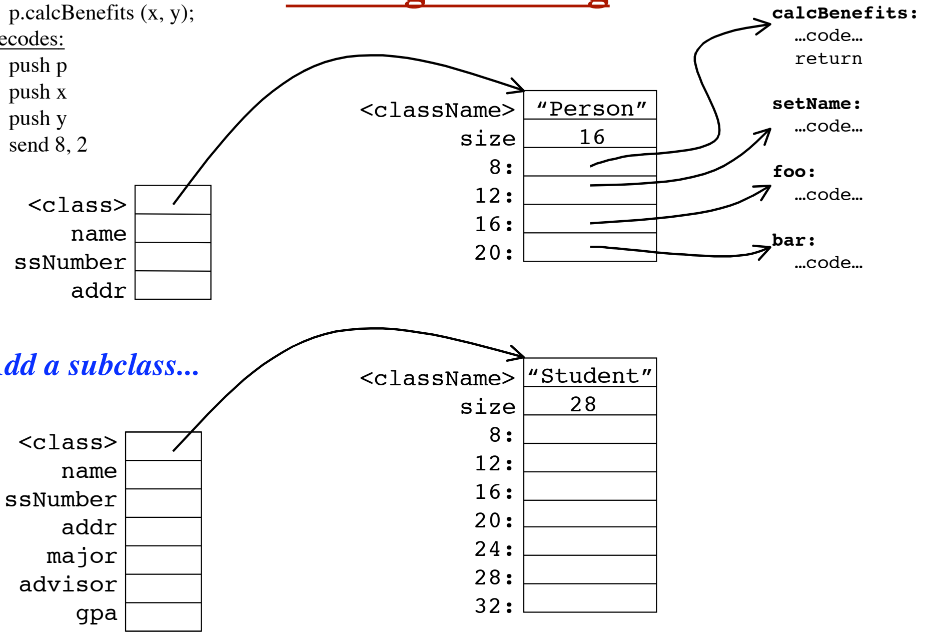
Message Sending

Source:

p.calcBenefits (x, y);

Bytecodes:

push p
push x
push y
send 8, 2



<class>
name
ssNumber
addr

<className> "Person"
size 16
8:
12:
16:
20:

calcBenefits:
…code…
return

setName:
…code…

foo:
…code…

bar:
…code…

Add a subclass...

<class>
name
ssNumber
addr
major
advisor
gpa

<className> "Student"
size 28
8:
12:
16:
20:
24:
28:
32:

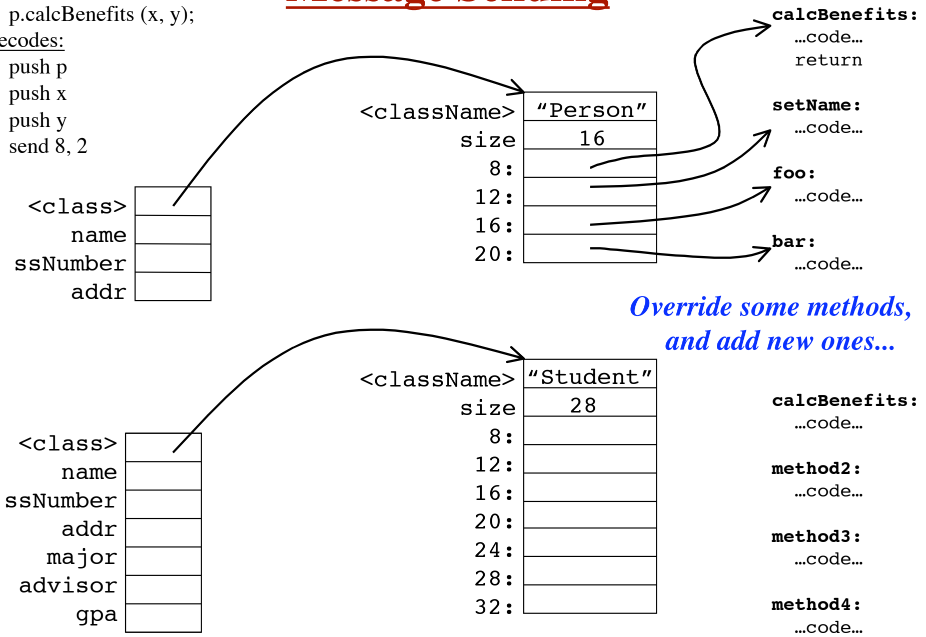
Message Sending

Source:

p.calcBenefits (x, y);

Bytecodes:

push p
push x
push y
send 8, 2



<class>
name
ssNumber
addr

<className> "Person"
size 16
8:
12:
16:
20:

calcBenefits:
…code…
return

setName:
…code…

foo:
…code…

bar:
…code…

Override some methods,
and add new ones...

<class>
name
ssNumber
addr
major
advisor
gpa

<className> "Student"
size 28
8:
12:
16:
20:
24:
28:
32:

calcBenefits:
…code…

method2:
…code…

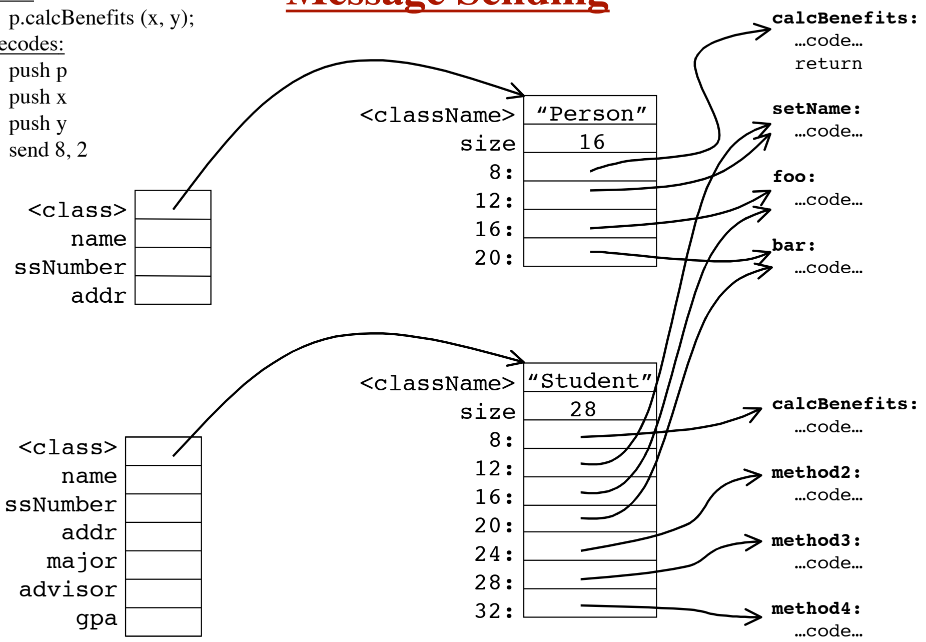
method3:
…code…

method4:
…code…

Message Sending

Source:
p.calcBenefits (x, y);

Bytecodes:
push p
push x
push y
send 8, 2



<class>
name
ssNumber
addr

<className> "Person"
size 16
8:
12:
16:
20:

calcBenefits:
…code…
return

setName:
…code…

foo:
…code…

bar:
…code…

<class>
name
ssNumber
addr
major
advisor
gpa

<className> "Student"
size 28
8:
12:
16:
20:
24:
28:
32:

calcBenefits:
…code…

method2:
…code…

method3:
…code…

method4:
…code…

Collection Classes

```
import java.util.*;
```

Interfaces	Classes
List	LinkedList
	ArrayList
Set	HashSet
Map	HashMap
	TreeMap
Collection	AbstractCollection

Example

```
List myList = new LinkedList ();
Person p = ...;
...
myList.add (p);
...
i = myList.size ();
if (myList.isEmpty ()) ...
```

How to go through the List?

“Iterators”

Class: Iterator
Methods: hasNext, next

```
Iterator it = myList.iterator ();
while (it.hasNext ()) {
      p = (Person) it.next ();
      ... Use p ...
}
```

Many Collections understand the iterator message.

An iterator is like a pointer into the collection.
    hasNext --> *Has the pointer reached the last element?*
    next --> *Get the next item and advance the pointer*.

Must not modify the underlying collection
    while an iterator is being used on it!

Iterators also understand the remove message.

When extracting an element, you must always *cast* the value.

```
p = (Person) it.next ();
```

Linked List Methods:

```
getFirst () ➔ Object
getLast () ➔ Object
addFirst (Object) ➔ Object
addLast (Object) ➔ Object
removeFirst (Object) ➔ Object
removeLast (Object) ➔ Object
```

Example: A Stack

```
LinkedList st = new LinkedList ();
...
st.addFirst (p);                   // Push
...
p = (Person) st.removeFirst ();    // Pop
```

What about Basic Types in Collections???

```
LinkedList myList = ...;

myList.addFirst (1257);
```

Can't convert "int" to Object!

```
i = (int) myList.removeLast ();
```

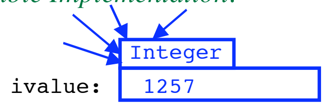
removeLast returns an Object.
Can't cast an Object to "int"!

"Wrapper" Classes

Basic Types	Corresponding "Wrapper" Classes
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
bool	Boolean

Spelling is similar to the basic type names

Possible Implementation:



```
class Integer {
  private int ivalue;
  Integer (int i) { ivalue = i; }
  int intValue () { return ivalue; }
}
```

List of Integers?

Basic Methods in Integer:

Constructors:

```
Integer (int) ➔ Integer
          Used to wrap a value in an object

Integer (String) ➔ Integer
          "Parse" the given string to get a value and wrap it
```

Methods:

```
intValue () ➔ int
          Used to extract the value

toString () ➔ String
          Used to get a printable version of the value

valueOf (String) ➔ int
          int i = Integer.valueOf ("1257");

equals (Object) ➔ bool
          Integer i,j = ...;
          if (i == j) ...          // OOPS!
          if (i.equals (j)) ...    // Okay
```

Example:

```
LinkedList myList = ...;

myList.addFirst (new Integer (1257));

i = ((Integer) myList.removeLast ()).intValue ();
```

Operations on Integer

Constructors:

```
Integer (int) ➔ Integer
          Used to wrap a value in an object

Integer (String) ➔ Integer
          "Parse" the given string to get a value and wrap it
```

Methods:

```
intValue () ➔ int
          Used to extract the value

toString () ➔ String
          Used to get a printable version of the value

valueOf (String) ➔ int


equals (Object) ➔ bool
```

Operations on Double

Constructors:

```
        Double (double) ➔ Double
                Used to wrap a value in an object

        Double (String) ➔ Double
                "Parse" the given string to get a value and wrap it
```

Methods:

```
        doubleValue () ➔ double
                Used to extract the value

        toString () ➔ String
                Used to get a printable version of the value

        valueOf (String) ➔ double


        equals (Object) ➔ bool
```

Etc, for the other wrapper classes...

Static Methods for Character

```
getNumericValue (char) ➔ int
         int i = Character.getNumericValue ('5');

digit (char c, int radix) ➔ int
         int i = Character.digit ('E', 16);  // Sets i to 14

forDigit (int i, int radix) ➔ char
         char x = Character.forDigit (11, 16);   // Sets x to 'b'

isDigit (char) ➔ bool
         if (Character.isDigit (x)) ...

isLetter (char) ➔ bool

isLetterOrDigit (char) ➔ bool

isWhiteSpace (char) ➔ bool

isUpperCase (char) ➔ bool

isLowerCase (char) ➔ bool

toUpperCase (char) ➔ char

toLowerCase (char) ➔ char
```

Static Methods for Integer

```
toHexString (int) ➔ String
        Integer.toHexString (127)  -->  returns “7f”
```

Double

Constants

```
    POSITIVE_INFINITY
    NEGATIVE_INFINITY
    NaN
          double d = Double.POSITIVE_INFINITY;
```

Static Methods

```
    isNan (double) ➔ bool
    isInfinite (double) ➔ bool
          if (Double.isInfinite (d)) ...
```

Instance Methods

```
    x.isNan () ➔ bool
    x.isInfinite () ➔ bool
```

Also for Float

Parsing Methods...

```
    parseDouble (String) ➔ double
          double d = Double.parseDouble (“3.1415”);
    parseFloat (String) ➔ float
    parseInt (String) ➔ int
    ...
```

Conversion to String Representation...

```
    toString (double) ➔ String
          String s = “value= “ + Double.toString (d);
    toString (float) ➔ String
    toString (int) ➔ String
    ...
```

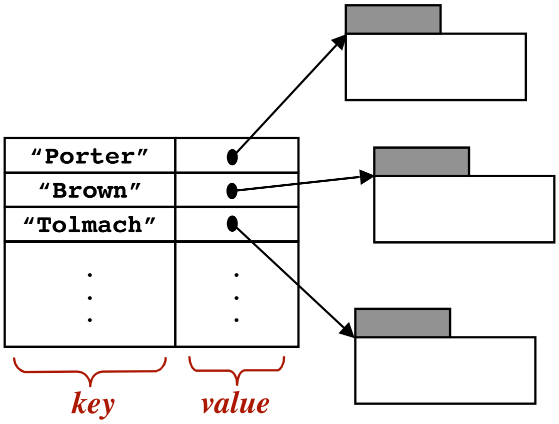
For all Wrapper Classes

Map

A look-up table
    A set of entries

Each entry has
    Key
    Value

Examples:
    Phonebook
    Dictionary

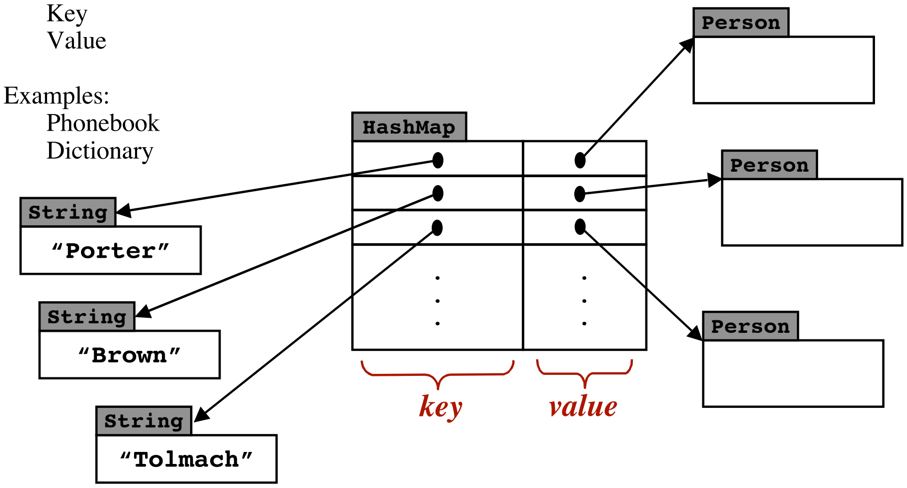


Map

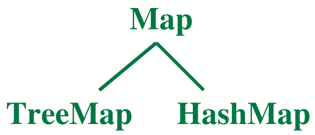
A look-up table
    A set of entries

Each entry has
    Key
    Value

Examples:
    Phonebook
    Dictionary



Map



```
Map m = new HashMap ();

  ...

m.put ("Tolmach", p1);
m.put ("Porter", p2);

  ...

p = (Person) m.get ("Porter");

  ...

if (m.containsKey ("Brown")) ...

  ...

m.remove ("Tolmach");
```

Replace value if key already there

Returns null if not there

The value may be "null" but be careful of confusion when "get" returns null!

Also returns the value (or null)