# Java Essentials

(including differences from C++)

## Java Features

Platform independent
- "Write once, run anywhere"

Object-oriented

Safe references
- Class casting checked at run-time
- No dangerous pointer manipulations

Garbage collection

Built-in exception handling

Support for multi-threading, networking, security, web applets, etc.

## Some Shortcomings

Rather slow compared to fully-compiled code
- Changing with on-the-fly compilation technology

Some unavoidable space inefficiencies
- No arrays of classes without references

Difficult to take advantage of platform-specific features

## Sun's Java Tools

javac : Java byte code compiler
- Compiles Java source to platform-independent byte codes

java : Java run-time environment
- Verifies byte codes for security correctness and executes on Java Virtual Machine

jdb : Java debugger

JSwat: Graphical debugging environment
- written in Java, for Java

Available free from http://www.bluemarsh.com

## Hello, World!

```
class HelloWorld {
  public static void main(String[] args)
      System.out.println("Hello, World!");
}
```

## Topics to Cover

Basic types

Operators

Flow Control

Classes

Inheritance

Interfaces

Error Handling

## Primitive Types

| | | |
|---|---|---|
| | boolean | true or false |
| | char | 16-bit Unicode character |
| integer types | byte | 8-bit signed integer |
| | short | 16-bit signed integer |
| | int | 32-bit signed integer |
| | long | 64-bit signed integer |
| float types | float | 32-bit floating point |
| | double | 64-bit floating point |

## Differences from C++

boolean true and false do not have integer equivalents

char is not a byte, but a 16-bit Unicode

No unsigned integer types (so byte goes to 128, not 256)

All types have specified size

## Strings

Strings are special built-in class

<string> + <string> performs concatenation

- creates new string that combines the two

String a = “foo”;

String b = “bar”;

String c = a + b + “!”

c.length == 7

## Variables

No global variables

May appear as Class fields or local

Appear as

[modifiers] <type> <variable-name> [= <val>]

Possible modifiers:

- final - unchangeable constant
- static - only one instance
- public, private, protected - access restrictions
- synchronized - for multithreading

## Operators and Precedence

```
[ ]  .  (params)
expr++  expr--   ++expr   --expr  +expr -expr   ~    !
new    (type)expr
*   /   %  (floats, integers)
+   -   (integers, floats, strings)
<<   >>   >>>   (integers only)
<   >   >=   <=   instanceof
== !=
&
^
|
&&  (booleans only)
||      (booleans only)
?:
=  +=  -=  *=  /=  %=  >>=  <<=  >>>=  &=  ^=  |=
```

Note: no operator overloading in Java

## Precedence Example

What is: 5 + 21 / 4 % 3

= (5 + ((21 / 4) % 3))

= 5 + ((5) % 3)

= 5 + (2)

= 7

# Explicit Casting

**Explicit**

- **(type)expression**
- **Possible among all integer and floating types**
- **Possible among some class references**
- **int i = (int)( (double)5 / (double)3 )**

# Implicit Casting

**Implicit**

- **Applied automatically when no information lost**
  - —float → double
  - —byte → short → int → long
  - —int → double
  - —double d = 6;  d = 7 / 2;   d = 7 / 2.0;
- **Any type converted to String when involved in String concatenation (+)**
  - —String s = 8 + " Days a Week"

# Control Flow - if/else

```
if (boolean)
  statement1;
else if (boolean)
  statement2;
else
  statement3;
```

***Booleans only, not integers***

- `if (i > 0)`      **legal**
- `if (i = x--)`    **illegal**

# Switch/case

```
switch (<integer>) {
  case <const 1>:
     statements;
     break;
  case <const 2>:
     statements;
     break;
  default:
     statements;
}
```

# Switch/case Example

What is printed?

```
int i = 3;
switch (i) {
  case 3:
      System.out.println("3");
  case 6:
      System.out.println("6");
  default:
      System.out.println("Default");
}
```

# Loops

```
while (<boolean>)
  statement;

do
  statement;
while (<boolean>)

for (init-expr; <boolean>; incr-expr)
  statement;
```

## Loop Refresher

Which loops must execute their statements at least once?

Which loops can choose to never execute their statements?

Which value of the boolean indicates to do the statements again?

Do you know a loop construct in C that is the opposite of this?

## Labelled Break and Continue

Break jumps out of enclosing switch or loop

Continue jumps to increment section of loop

Labels may be placed before switch or loop to determine which is indicated by break or continue

```
foo:
while (<boolean>)
  while (<boolean>)
    ...
    break foo;
```

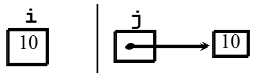
## Continue Example

To where does execution jump after the continue is executed?

```
Zen:
 for (int i=0; i<10; i++) {
    for (int j=0; j<i; j++) {
        if (i+j == 5)
            continue Zen;
    }
 }
```

## Value vs. Reference Variables

Value | Reference



```
int i=10;
```

```
int[] j = new int[1];
j[0] = 10;
```

Or `int[] j = {10};`

**new** performs dynamic memory allocation

Variables of primitive types are value variables

Variables of arrays and classes are reference variables

- Reference variables are a safer form of pointers

(In C++, you can choose)

## Passing Parameters

All variables passed "by value"

- Contents of variable are copied to variable inside the procedure

```
foo(int i, int[] ia1, int[] ia2) {
  i--;
  ia1[0] = 6;
  ia2 = ia1;
}
int i=1;
int[] array1={3}, array2={4};
foo(i, array1, array2);
```

What are the values of `i`, `array1[0]`, and `array2[0]` after returning from `foo()`?

## Arrays

Refer to several values of same type

Example:

```
int[] myArray = new int[20];
```

Length field holds allocated number of elements

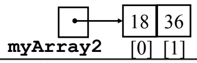
```
myArray.length == 20
```

Indexed from 0..length-1

- bounds checked dynamically

Initialized manually or in declaration

```
int[] myArray2 = {18, 36};
```



# 2D (and higher D) Arrays

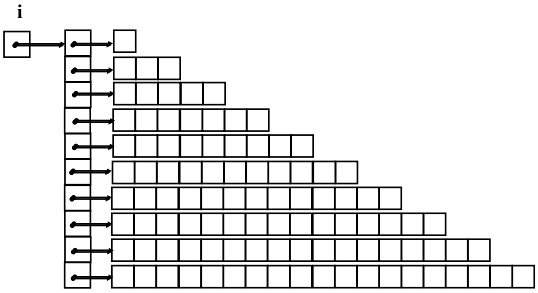
**May be allocated at once for rectangles**

```
int[][] i = new int[12][15];
```

**Or deal with 1 dimension at a time**

```
int[][] i = new int[10][];
for (int j=0; j<10; j++)
  i[j] = new int[2*j + 1];
```

# Non-rectangular 2D Array



# Classes

**Combine *fields* (variables) and *methods* (procedures)**
**Fields and methods accessed by . (dot) operator**

```
class MyClass {
  static int numInstances=0;
  protected int somethingImportant;
  public int tellAll() {
    return somethingImportant;
  }
}

MyClass myVar = new MyClass;
System.out.println(MyClass.numInstances +
  “, “ + myVar.tellAll());
```

# Fields

**Modifiers**

- **public, protected, private : affect visibility**
- **static : affects instantiation**
- **final : makes field a constant**
- **synchronized : used for multithreading**

# Field Instantiation

**Class fields**

- **static - only one per class**
- **May be accessed without a class variable**
  - **—<classname>.<static field>**
    - **» e.g. Math.PI**

**Instance fields**

- **non-static - one per class instance**

# Field and Method Visibility

**public, protected, private, or “package” (default)**

| Accessible to: | public | protected | package | private |
|---|---|---|---|---|
| same class | yes | yes | yes | yes |
| class in same package | yes | yes | yes | no |
| subclass in different package | yes | yes | no | no |
| non-subclass, different package | yes | no | no | no |

## Methods

Modifiers

- Same as fields, plus abstract

May be overloaded (methods with same name)

- Must have different signatures (defined by parameter type sequence)

`static` methods cannot access instance variables

`this` provides reference to class instance

- Can be passed as parameter to another method
- Can disambiguate class fields from parameters

## Initialization/Constructors

Initialization performed when class is *instantiated* by:
`new <class>[(params)]`

- Fields initialized to specified values or to defaults according to type
- Constructor called if there is one (params must match a constructor signature)
- Constructors may be overloaded as well

```
MyClass( ) { numInstances++;
 somethingImportant = numInstances*3; }
```

## `finalize()` method

If defined, called during garbage collection

Not necessary for deallocating space

Sometimes useful for freeing up other resources

- closing files, etc.

## Inheritance

Enables class extensions with reuse of some fields and methods

- All parent fields included in child instantiation
- Protected and public fields and methods directly accessible to child
- Parent methods may be *overridden*
- New fields and methods may be added to child
- Only single inheritance (unlike C++)

## Simple Inheritance Example

```
Class MyExtension extends MyClass {
  float newField;
  MyExtension() {
    super(); //call parent constructor
    newField = super.tellAll()*3.14;}
  int tellAll(){return (int)newField;}
}

MyClass foo = new MyExtension;
System.out.println(foo.tellAll());
```

Method accessed is that of actual instantiation type, not variable type

- In C++ terminology, all functions are “virtual”

## Initialization of Derived Classes

`super()`: alias for constructor of parent class

- Constructor of derived class can explicitly call `super()` (with or without arguments) to invoke parent constructor
- If constructor does not call `super()` or `this()` at start of constructor
  - `super()` is automatically called (with no arguments)
- Inside `super()` function, `this` object has been cast to parent class

## Casting of Class Variables

"upward" casting

- Casting derived class variable to ancestor class is always safe (and may be done implicitly)

"downward" casting

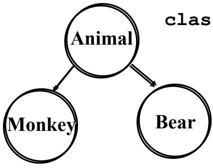
- Casting class variable to derived class fails if variable is not actually an instance of the derived class

  —run-time error

- `instanceof()` operator can be used to test class type before downward cast

## Class Casting Example

```
class Animal {...}
class Bear extends Animal {...}
class Monkey extends Animal {...}
```



```
Animal a;
Monkey m;
Bear b;
```

```
a = new Bear();  // legal
b = (Bear)a;     // legal
m = (Monkey)a;   // illegal
m = (Monkey)b;   // illegal
```

## Abstract Classes

Include one or more unimplemented abstract methods

Enable inheritance of methods that don't make sense at the parent level

```
abstract class Shape {
  public int id;
  abstract public draw(); }

class Circle extends Shape {
  public draw() {...} }

class Rectangle extends Shape {
  public draw() {...} }

Shape s = new Circle();
s.draw();
```

## Interfaces

```
interface Printable {
  print(); }
class Foo implements Printable {
  print(){...}; }
```

Similar to abstract classes

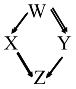
- But *no* methods implemented or fields specified

Class can be defined to *implement* one or more interfaces

- More general mechanism than just single inheritance

Variables may actually use interface as type

## "Multiple Inheritance"



Several ways to achieve in Java

- combinations of interfaces and classes
- W and Y are interfaces, X and Z are classes
- W, X, and Y are interfaces, Z is class

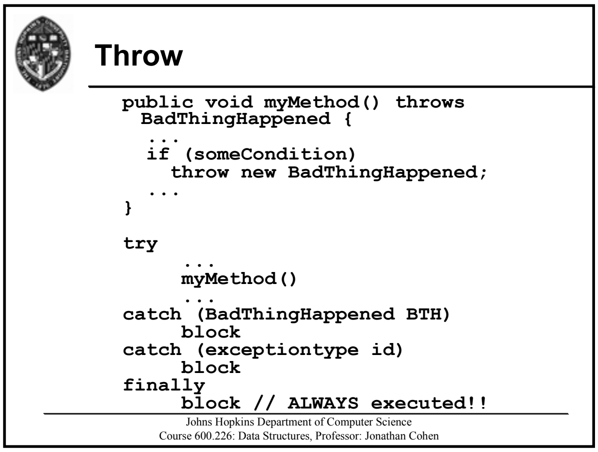
## Exceptions

Language-level support for managing run-time errors

You can define your own exception classes

Methods declare which exceptions they might possibly *throw*

Calling methods either *catch* these exceptions or pass them up the call stack



# Throw

```
public void myMethod() throws
  BadThingHappened {
   ...
   if (someCondition)
     throw new BadThingHappened;
   ...
}

try
      ...
      myMethod()
      ...
catch (BadThingHappened BTH)
      block
catch (exceptiontype id)
      block
finally
      block // ALWAYS executed!!
```