The Joy of Coding

The Java™ programming platform consists of the Java programming language, a set of standard libraries, and the virtual machine in which Java programs run. The syntax of the Java programming language closely resembles C, but is object-oriented and has features such as a built-in boolean type, support for international character sets, and automatic memory management.

The Java Programming Language

- What is Java?
- The Java Programming Language
- Object-oriented programming
- Tools for compiling and running Java

What is Java?

"A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multi-threaded, and dynamic language."
– Original Java White Paper

"C++ without the guns, knives or clubs."
– James Gosling

Java is a programming platform for developing portable, hardware-independent applications and libraries.

Java provides a standard application programming interface (API) for dealing with common data structures (e.g. linked lists and hash tables), file I/O, etc.

Java was designed with modern computing in mind and thus contains facilities for networking, graphics, and security.

Java is a proven language and platform for application and enterprise-level computing

Java development tools can be downloaded from:

`https://openjdk.org`

The Java Programming Language

Developed by Ken Arnold and James Gosling at Sun Microsystems

Java programs are built from *classes* that have *methods* consisting of *statements* that perform work.

Every class is in a *package*. Packages provide a naming context for classes.

Program execution begins with a `main` method whose sole argument is an array of `String`s that are the command line arguments.

The infamous Hello World program:

```
package edu.pdx.cs.joy.lang;

public class Hello {   // Hello is a class
  // main is a method
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

This program prints the string `Hello World` to standard output.

Java's Execution Environment

Java was designed to be platform independent

"Write once, run anywhere"

Java programs (classes) are compiled into binary *class files* that contain machine instructions called *bytecodes*.

The bytecodes are executed by a Java Virtual Machine (JVM).

JVMs are platform-dependent and are usually implemented in a language like C.

Compiling a Java Program

A Java class is described in a text file that contains its source code

The name of the file corresponds to the name of the class

It is a good idea to place source files in a directory whose name corresponds to the class's package

For instance, our Hello World class's source file is:

`edu/pdx/cs/joy/lang/Hello.java`

These conventions help keep your source code organized

The `javac` tool compiles Java source code into bytecode that is placed in a *class file*

```
$ cd edu/pdx/cs/joy/lang
$ javac -d ~/classes Hello.java
```

The `-d` option specifies where the class file is written

`~/classes/edu/pdx/cs/joy/lang/Hello.class`

Executing a Java Program

The `java` command invokes the Java Virtual Machine

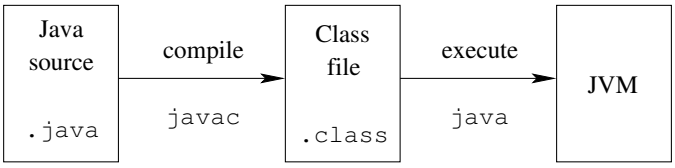
Execution begins with the `main` method of the given class. (Note that `java` takes the name of a **class**, not the name of a class file!)

`java` requires the fully-qualified (including package) name of the `main` class

You also need to tell the JVM where to look for class files

- The `-classpath` (or `-cp`) option specifies a directory path to search for classes
- Alternatively, you can set the `CLASSPATH` environment variable

```
$ java -cp ~/classes edu.pdx.cs.joy.lang.Hello
Hello World
```



Types in the Java Programming Language

In addition to `class` types, Java has eight *primitive types*:

`boolean`	`true` or `false`
`char`	16-bit Unicode 1.1 character
`byte`	8-bit signed integer
`short`	16-bit signed integer
`int`	32-bit signed integer
`long`	64-bit signed integer
`float`	32-bit IEEE 754-1985 floating point
`double`	64-bit IEEE 754-1985 floating point

Literal values for Java primitives are like C

- `int` literals: `42`, `052`, `0x2a`, `0X2A`, `1_996`
- `long` literals: An `int` literal with `L` or `l` appended
- `char` literals are delimited by single quotes: `'q'`, `''`
- `String` literals* are delimited by double quotes: `"Hello"`

*are not the same as `char` arrays!

Comments in Java Programs

Java allows both C* and C++ style comments.

```
package edu.pdx.cs.joy.lang;

/**
 * This program prints out the product of two
 * <code>double</code>s.
 */
public class Product {
  public static void main(String[] args) {
    /* Multiply two doubles... */
    double product = 3.5 * 1.8;
    System.out.println(product);  // Print product
  }
}
```

Comments between `/**` and `*/` are called *documentation comments*.

Doc comments describe the class or method directly following them.

The `javadoc` tool uses these comments to generate HTML documentation for classes.

*Java comments do not nest!

Control Flow

Java has all of the standard control flow mechanisms you would expect from a modern programming language

- `if`, `else`, `for`, `while`, `do while`, `continue`, `break`

Note that Java does not have a `goto` operation

- The `goto` keyword is reserved to mean nothing

Java also has "enhanced" `for` loop syntax that doesn't require an index variable:

```
int[] array = ...
int sum = 0;
for (int i : array) {
  sum += i;
}
```

This syntax can be read as:

> "For each `int`, `i`, in array, `array`, do..."

Exceptions: When Bad things happen to Good Objects

Errors of varying severity may occur during program execution

- Dividing by zero
- Trying to read a non-existent file
- Indexing beyond the end of an array

If your program tried to accommodate every potential error case, it would become extremely messy.

Java uses *exceptions* to signal errors without cluttering code

When an error condition occurs, an exception is *thrown*. Methods *declare* that they might throw an exception.

An exception is *caught* by encompassing code and handled appropriately (e.g. printing an error message or prompting the user to enter another file name).

Kinds of exceptions

Checked exceptions are unexpected, but not surprising (e.g. a file cannot be found).

If a method may throw a checked exception, it must be declared in the method's `throws` clause.

```
public AppData readFromFile(File file)
  throws FileNotFoundException {

  // Read from the file and create an AppData object}
```

Unchecked exceptions are more rare and usually signal a serious problem with your program (e.g. a divide by zero, there is no more memory left).

- Unchecked exceptions are subclasses of `java.lang.RuntimeException` or `java.lang.Error`

Catching exceptions

Exceptions are handled with a `try-catch-finally` block

- A `catch` block can catch multiple types of exceptions

```
try {
  // Code that may throw an exception or two
  AppData data = readFromFile(file);
  writeToDatabase(dbConnection, data);

} catch (FileNotFoundException | SQLException ex) {
  // Executed if the exception was throw
  printOutErrorMessage(ex.getMessage());

} catch (EndOfFileException ex) {
  // Some exceptions can be ignored

} finally {
  // Executed regardless of whether or not an
  // exception was thrown ("clean up code")
  closeFileStream(file);
  closeDatabaseConnection(dbConnection);
}
```

Packages

Naming conflicts often arise when code is reused

- Multiple companies have a `User` class

Java uses packages to distinguish between classes

- Every class is in a package
- Classes in the same package have related functionality (e.g. `java.net`)
- Packages are hierarchical in nature
  - The standard Java packages begin with `java`
  - The classes for this course begin with `edu.pdx.cs.joy`
  - All of your classes must begin with `package edu.pdx.cs.joy.`*userid*
- A class's package is specified by the `package` declaration at the top of its source file
  - If no `package` declaration is specified the class is placed in the *default package*

Packages

When a class references another class that is outside of its package, it must be *fully-qualified* or `imported`

Classes in the `java.lang` package are implicitly imported

```
package edu.pdx.cs.joy.lang;

import java.io.File;  // Just File from java.io

public class Packages {
  public static void main(String[] args) {
    File file = new File(args[0]);
    java.net.URL url = file.toURL();
    String name = url.toString();
    System.out.println(name);
  }
}
```

You can import all of the classes in a package with `import java.io.*;`

Note that the `*` is not recursive!

- `import java.*` will not import all Java classes

Differences Between Java and C++

Java does not have `goto`

(Checked) Exceptions in Java must be caught

No "pass by reference" (no pointers, either)

No << operator for I/O

Java operators cannot be overridden nor overloaded by the programmer

No preprocessor (`#define` and friends)

The `!` operator can only be used with `boolean`s

- Illegal code:

```
void wrong(int bad) {
  if (!bad) {
    // ...
  }
}
```


The `jar` Tool

`jar` is a tool that allows you to combine multiple files (usually class files) into a "Java Archive"

Let's you put related classes (e.g. all of the classes in your library or application) into a single file

- Makes downloading easier
- Jar files can be "signed" for security

`jar` looks a lot like the UNIX `tar` tool:

```
$ cd ~/classes
$ jar cf classes.jar .
$ jar tf classes.jar
META-INF/
META-INF/MANIFEST.MF
edu/
edu/pdx/
edu/pdx/cs/joy/
edu/pdx/cs/joy/lang/
edu/pdx/cs/joy/lang/Hello.class
```

The `jar` Tool

The JVM knows how to read classes from a jar file

```
$ java -cp classes.jar edu.pdx.cs.joy.lang.Hello
Hello World
```

We will be using a number of jar files:

`projects.jar`	Classes for your project
`examples.jar`	Example code for this course
`family.jar`	The family tree application
`grader.jar`	Code for submitting your projects

Jar files may have their contents compressed and may contain files such as pictures and sounds that may be needed by an application

A jar file contains a special entry called the *manifest* that describes the jar file

- Version, creator, digital signature information, etc.

Classes in the `java.util.jar` package can be used to manipulate jar files

Documenting Your Code with `javadoc`

`javadoc` is a utility that generates HTML documentation for Java classes

The HTML contains links between classes for easy navigation

`javadoc` uses documentation comments that describe classes, methods, and exceptions

Documentation is placed between `/**` and `*/` comments that precede the class, field, or method being described

Special comment tags used with `javadoc`:

`@author`	Who wrote it
`@param`	Description of a parameter to a method
`@return`	Description of value returned by a method
`@throws`	An exception thrown by the method
`@see`	References another class or method

A class with `javadoc` comments

```
package edu.pdx.cs.joy.lang;

/**
 * This class demonstrates Javadoc comments
 *
 * @author David Whitlock
 * @version 1.0
 */
public class Javadoc {

  /**
   * Returns the inverse of a <code>double</code>
   * @param d
   *        The <code>double</code> to invert
   * @return The inverse of a <code>double</code>
   * @throws IllegalArgumentException
   *         The <code>double</code> is zero
   */
  public double invert(double d)
   throws IllegalArgumentException {
    if (d == 0.0) {
      String s = d + " can't be zero!";
      throw new IllegalArgumentException(s);
    } else {
      return 1.0 / d;
    }
  }
}
```


Using the `javadoc` tool

`javadoc` needs to know both the location of your source files and your class files

`-sourcepath` dir	Where to find Java source files
`-classpath` dir	Where to find class files
`-d` dir	Destination directory, where html files are placed

To generate HTML file for the `Javadoc` class (\ is the UNIX command line continuation character):

```
$ javadoc -classpath ~/classes -sourcepath src \
  -d ~/docs edu.pdx.cs.joy.lang.Javadoc
```

The HTML files will be placed in `~/docs`

Javadoc offers a powerful and standard means for documenting Java classes

Summary

Java programs are written with classes that contain methods. Execution starts with the `main` method.

Java programs are compiled into a intermediate binary form called bytecode

The bytecode is executed by a virtual machine thus making Java programs platform-independent

Java has many of the same primitive data types and control flow structures as C

To encourage good error handling, Java has built-in exceptions

Java classes are organized into packages

The `jar` tool is used to bundle class files together and the `javadoc` tool creates HTML documentation from Java source code

Object-Oriented Programming

The fundamental unit of programming in Java is the *class*

Classes contain *methods* that perform work

Classes may be *instantiated* to create *objects*; an object is an *instance* of a class

Object-oriented programming separates the notion of "what" is to be done from "how" it is done.

- "What": A class's methods provide a contract via their signatures (i.e. method's parameters types) and their semantics
- "How": Each class may have its own unique implementation of a method

When a method is invoked on an object, its class is examined at runtime to locate the exact code to run ("dynamic dispatch" or "virtual function")

Fields

A class's variables are called its *fields* of which there are two kinds

Instance fields are associated with an object. Each instance has its own value of the field.

Class fields are associated with the class itself. All instances of a class share the same value of a class field. Class fields are denoted by using the `static` keyword.

By convention, class names begin with a capital letter (`Employee`), while field and method names begin with a lowercase letter (`name`)

Access Control

Fields and methods (class "members") are always available to their declaring class, but you can control other class's access them with *access control modifiers:*

- *public:* Members declared `public` are accessible anywhere the class is accessible
- *private:* `private` members are only accessible by the class in which they are declared
- *protected:* `protected` members are accessible from direct subclasses and by classes in the same package
- *package:* Members with no declared modifier (default) are accessible only by classes in the same package

Constructors

Constructors are pseudo-methods that initialize a newly-created object. They are invoked when an object is instantiated using the `new` operator.

- Constructors have the same name as the class whose instances they initialize
- A constructor may have parameters, but it has no declared return type
- Instantiation allocates memory for an object in the JVM's garbage-collected heap

Methods

Methods contain code that often manipulates an object's state or perform an operation on behalf of an object

- Methods have *parameters* that are objects of their own type.
- Method also declare the type of data that they return (or `void` if they do not return data).

Methods are *invoked* on references to objects using the `.` operator:

receiver`.`*method*`(`*parameters*`)`

```
LocalDate now = LocalDate.now();   // class method
LocalDate then = now.minusDays(1); // instance method
```


Accessing `private` Data Using Methods

`public` fields are usually not a good idea. (Expose behavior, not state.) Have methods do the work.

You can use methods to control how fields are accessed

```
public class Employee {
  private int id;
  private String name;
  private Employee boss = null;

  private static int nextId = 0;

  public Employee(String name) {
    this.id = nextId++;
    this.name = name;
  }

  public void setBoss(Employee boss) {
    this.boss = boss;
  }

  public Employee getBoss() {
    return this.boss;
  }
}
```

`getBoss` and `setBoss` are called *accessor* and *mutator* methods, respectively.

Overloading Methods

A method's *signature* is comprised of the method's number and types of parameters

Method *overloading* involves having multiple methods with the same name, but different signatures

```
public void setName(String name) {
  this.name = name;
}

pubic void setName(String firstName,
                   String lastName) {
  this.name = firstName + " " + lastName;
}
```

The `setName` method is *overloaded*

Static Fields

A *static* member is associated with a class instead of an object

For instance, there is only one `nextId` variable for all `Employee`s (like a "global variable")

A class's static fields are initialized before any static field is referenced or any method is run

Static fields may also be initialized in a *static initialization block*

```
public class Day {
  public static String[] daysOfWeek;

  static {
    daysOfWeek = new String[7];
    daysOfWeek[0] = "Sunday";
    daysOfWeek[1] = "Monday";
    daysOfWeek[2] = "Tuesday";
    // ...
  }
}
```

```
  String monday = Day.daysOfWeek[1];
```


Static Methods

A static method is invoked with respect to an entire class, not just an instance of a class

Static methods are also called *class methods*

Static methods have no `this` variable, so static methods can only access static variables and can only invoke static methods of its declaring class

You usually invoke a class method with

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

```
public static String getDay(int i) {
  return daysOfWeek[i];
}
```

```
  String monday = Day.getDay(1);
```

Recall that the `main` method is `static`

Static import

You can use the `import static` statement to import all of the static members of a class:

```
package edu.pdx.cs.joy.j2se15;

import static java.lang.Integer.*;
import static java.lang.System.out;

public class StaticImports {

  public static void main(String[] args) {
    int sum = 0;
    for (int i = 0; i < args.length; i++) {
                          // Integer.parseInt()
      sum += parseInt(args[i]);
    }

                          // System.out
    out.println("Sum is " + sum);
                          // Integer.MAX_VALUE
    out.println("MAX_INT is " + MAX_VALUE);
  }
}
```


Memory Management

Objects in Java are explicitly allocated using `new`, but are implicitly deallocated by the "garbage collector" that runs in the background of a JVM

An object is considered garbage when it meets all of the following criteria

- It is not referenced by a static field
- It is not referenced by a variable in a currently executing method
- It is not referenced by a *live* (non-garbage) object

The programmer cannot destroy an object – there is no "delete" nor "free"

- No dangling references
- Makes the language a lot safer
- You can still have "memory leaks" (if objects remain live despite never needing to be used again)

Extending Classes

One of the main advantages of object-oriented programming is code reuse

One way to reuse classes is to extend their functionality by subclassing them

A class's methods and public fields specify a "contract" that it provides

When you *extend* a class, you add new functionality to its contract and you also *inherit* its existing contract

A subclass may change the implementation of its superclass's contract, but the semantics of the contract should not change

A subclass can be used wherever its superclass can be used. This property is called *polymorphism*.

`java.lang.Object` is the root of Java's class hierarchy

Everything "is an" `Object`

```
String s = "I'm a String!";
Object o = s;
```


`final` Classes and Methods

Marking a method as `final` prevents any subclass from overriding it

Marking a class as `final` prevents it from being extended

Final classes and methods provide security in that they ensure that behavior will not change

- A class is prevented from overriding a `final` `validatePassword` method to always return `true`

`final` Fields

The value of fields that are declared to be `final` cannot be changed

- Can only be assigned to in constructors
- A field that is `final` and `static` is a considered to be *constant*

```
public class Circle {
  public static final double PI = 3.14159;
  private final double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  public double getCircumference() {
    return 2.0 * PI * this.radius;
  }
}
```

The names of constants are usually in `ALL_CAPS`

Methods Provided by the `Object` Class

Every class inherits the following methods from `Object`

`public boolean equals(Object obj)`

- Compares the receiver object to another object
- Value equality (as opposed to == and `!=`)

`public int hashCode()`

- Returns a hash code for the object (used when storing the object in hash tables)

`protected Object clone() throws CloneNotSupportedException`

- Returns a copy (a "clone") of the receiver object

`public final Class getClass()`

- Returns the instance of `java.lang.Class` that represents the class of the receiver object

`protected void finalize() throws Throwable`

`abstract` Classes and Methods

Abstract classes allow you to delegate the implementation of a class or method to the subclass.

Makes sense when some behavior is true for most instances, but other behavior is specific to a given subclass.

Abstract methods are denoted with the `abstract` keyword and have no method body.

Abstract classes are denoted with the `abstract` keyword.

- Abstract classes cannot be instantiated, although they may declare constructors
- Any class that has an `abstract` method, must also be declared `abstract`

Interfaces

An *interface* is like a class, but it contains only method declarations*

- Used to declare methods that a class should implement
- Does not specify how those methods are implemented
- A class uses the `implements` keyword to denote that it implements the methods of an interface
- A class may implement more than one interface: multiple inheritance of behavior
- An interface may `extend` another interface
- All methods of an interface are implicitly `public`
- All fields of an interface are implicitly `public`, `static`, and `final` (constants)
- An interface is a type just like a class (i.e. you can have variables of an interface type)

*Interfaces may have `default` methods with implementations

Java Annotations

Java's "annotations" synatx for classes, fields, and methods provide additional information to tools and frameworks

- Annotations look like Javadocs, but they don't appear inside comments
- There are several built-in annotations in the `java.lang` package:
  - `@Deprecated`: A progam element is considered dangerous and should no longer be used
  - `@Override`: A method overrides a method of its superclass
  - `@SuppressWarnings`: Compiler warnings for the element (or its child elements) should not be issued
- An Annotation may have "elements" that configure its behavior
  - `@SuppressWarnings` has a `value` element that specifies which warnings are suppressed

Annotations Example

```
package edu.pdx.cs.joy.lang;

import java.util.Date;

/**
 * Demonstrates several of the built-in annotations
 *
 * @deprecated This class shouldn't be used anymore
 */
@Deprecated
public class AnnotationsExample {

  private final Date now = new Date();

  @SuppressWarnings({"deprecation"})
  @Override
  public boolean equals(Object o) {
    if (o instanceof AnnotationsExample) {
      AnnotationsExample other =
        (AnnotationsExample) o;
      return now.getDay() == other.now.getDay() &&
        now.getMonth() == other.now.getMonth() &&
        now.getYear() == other.now.getYear();
    }

    return false;
  }
}
```

Differences Between Java and C++

Java does not have pointers! You can't perform pointer arithmetic.

Java uses some different terminology

- "Methods" instead of "member functions"
- "Fields" instead of "member variables"
- "Instance methods" instead of "virtual functions"

Java does not have template classes (yet)

By default, instance methods are virtual

Java objects cannot be allocated on the runtime stack

Java does not have multiple class inheritance

No "friend" classes

No explicit memory deallocation

`!` operator only works on `booleans`

A class's declaration and definition are in the same file (no "header files")

All classes inherit from `Object`

Summary

Object-oriented programming separates data's interface from its implementation

Java classes have fields that hold data and methods that perform work

Classes, fields, and methods have access modifiers that encapsulate data and functionality

Static members are associated with classes and instance members are associated with objects

A class's functionality is extended through inheritance and by overriding its methods