Introducing Java

CSC207 Winter 2017

Reference Material on Java

See:

- the Lectures page for readings
- the Software page for a larger list of references

This reference is particularly useful:

http://docs.oracle.com/javase/tutorial/java/TOC.html

This website does a nice job walking you through Java:

https://www.sololearn.com/Course/Java/

Email registration is required. (Have you heard about disposable email addresses? Here's a top-15 article about the topic.)

Running Programs

What is a program?

What does it mean to “run” a program?

To run a program, it must be translated from the high-level programming language it is written in to a low-level machine language whose instructions can be executed.

Roughly, two flavours of translation:

- Interpretation
- Compilation

Interpreted vs. Compiled

Interpreted:

- e.g., Python
- Translate and execute one statement at a time

Compiled:

- e.g., C
- Translate the entire program (once), then execute (any number of times)

Hybrid:

- e.g., Java
- Translate to something intermediate (in Java, bytecode)
- Java Virtual Machine (JVM) runs this intermediate code

Compiling Java

You need to compile, then run (as described above).

If using command line, you need to do this manually.

First, compile using "javac":

```
dianeh@laptop$ javac HelloWorld.java
```

This produces file "HelloWord.class":

```
dianeh@laptop$ ls
HelloWorld.class  HelloWorld.java
```

Now, run the program using "java":

```
dianeh@laptop$ java HelloWorld
Hello world!
```

Most modern IDEs offer to do this for you (Eclipse does).

But you should know what's happening under the hood!

Defining Classes in Java

CSC207 Winter 2017

Instance Variables

```
public class Circle {
    private String radius;
}
```

`radius` is an instance variable. Each object/instance of the `Circle` class has its own `radius` variable.

Constructors

A constructor has:

- the same name as the class
- no return type (not even `void`)

A class can have multiple constructors, as long as their signatures are different.

If you define no constructors, the compiler supplies one with no parameters and no body.

If you define any constructor for a class, the compiler will no longer supply the default constructor.

`this`

`this` is an instance variable that you get without declaring it.

It's like `self` in Python.

Its value is the address of the object whose method has been called.

Defining methods

- A method must have a return type declared. Use `void` if nothing is returned.
- The form of a return statement:

  `return` *expression;*

  If the expression is omitted or if the end of the method is reached without executing a return statement, nothing is returned.
- Must specify the accessibility. For now:

  `public` – callable from anywhere

  `private` – callable only from this class
- Variables declared in a method are local to that method.

Parameters

When passing an argument to a method, you pass what's in the variable's box:

- For class types, you are passing a reference. (Like in Python.)
- For primitive types, you are passing a value. (Python can't do anything like this.)

This has important implications!

You must be aware of whether you are passing a primitive or object.

Instance Variables and Accessibility

If an instance variable is private, how can client code use it?

Why not make everything public — so much easier!

Encapsulation

Think of your class as providing an abstraction, or a service.

- We provide access to information through a well-defined interface: the public methods of the class.
- We hide the implementation details.

What is the advantage of this "encapsulation"?

- We can change the implementation — to improve speed, reliability, or readability — and *no other code has to change*.

Conventions

Make all non-final instance variables either:

- *private*: accessible only within the class, or
- *protected*: accessible only within the package.

When desired, give outside access using “getter” and “setter” methods.

[A *final* variable cannot change value; it is a constant.]

Access Modifiers

Classes can be declared public or package-private.

Members of classes can be declared public, protected, package-protected, or private.

Modifier	Class	Package	Subclass	World
`public`	Yes	Yes	Yes	Yes
`protected`	Yes	Yes	Yes	No
`default (package private)`	Yes	Yes	No	No
`private`	Yes	No	No	No

Inheritance in Java

CSC207 Winter 2017

Inheritance hierarchy

All classes form a tree called the inheritance hierarchy, with `Object` at the root.

Class `Object` does not have a parent. All other Java classes have one parent.

If a class has no parent declared, it is a child of class `Object`.

A parent class can have multiple child classes.

Class `Object` guarantees that every class inherits methods `toString`, `equals`, and others.

Inheritance

Inheritance allows one class to inherit the data and methods of another class.

In a subclass, `super` refers to the part of the object defined by the parent class.

- Use `super.`*`«attribute»`* to refer to an attribute (data member or method) in the parent class.
- Use `super(`*`«arguments»`*`)` to call a constructor defined in the parent class.

Constructors and inheritance

If the first step of a constructor is `super(«arguments»)`, the appropriate constructor in the parent class is called.

- Otherwise, the no-argument constructor in the parent is called.

Net effect on order if, say, A is parent of B is parent of C?

Which constructor should do what? Good practise:

- Initialize your own variables.
- Count on ancestors to take care of theirs.

Multi-part objects

Suppose class `Child` extends class `Parent`.

An instance of `Child` has

- a `Child` part, with all the data members and methods of `Child`
- a `Parent` part, with all the data members and methods of `Parent`
- a `Grandparent` part, … etc., all the way up to `Object`.

An instance of `Child` can be used anywhere that a `Parent` is legal.

- But not the other way around.

Name lookup

A subclass can reuse a name already used for an inherited data member or method.

Example: class `Person` could have a data member `motto` and so could class `Student`. Or they could both have a method with the signature `sing()`.

When we construct

```
x = new Student();
```

the object has a `Student` part and a `Person` part.

If we say `x.motto` or `x.sing()`, we need to know which one we'll get!

In other words, we need to know how Java will look up the name `motto` or `sing` inside a `Student` object.

Name lookup rules

For a method call: expression.method(arguments)

- Java looks for method in the most specific, or bottom-most part of the object referred to by expression.
- If it's not defined there, Java looks "upward" until it's found (else it's an error).

For a reference to an instance variable: expression.variable

- Java determines the type of expression, and looks in that box.
- If it's not defined there, Java looks "upward" until it's found (else it's an error).

Shadowing and Overriding

Suppose class `A` and its subclass `AChild` each have an instance variable `x` and an instance method `m`.

`A`'s `m` is **overridden** by `Achild`'s `m`.

- This is often a good idea. We often want to specialize behaviour in a subclass.

`A`'s `x` is **shadowed** by `Achild`'s `x`.

- This is confusing and rarely a good idea.

If a method must not be overridden in a descendant, declare it `final`.

Casting for the compiler

If we could run this code, Java would find the `charAt` method in `o`, since it refers to a `String` object:

```
Object o = new String("hello");
char c = o.charAt(1);
```

But the code won't compile because the compiler cannot be sure it will find the `charAt` method in `o`.

Remember: the compiler doesn't run the code. It can only look at the type of `o`.

So we need to cast `o` as a `String`:

```
char c = ((String) o).charAt(1);
```

Javadoc

Like a Python docstring, but more structured, and placed above the method.

```
/**
 * Replace a square wheel of diagonal diag with a round wheel of
 * diameter diam. If either dimension is negative, use a wooden tire.
 * @param diag  Size of the square wheel.
 * @param diam  Size of the round wheel.
 * @throws PiException  If pi is not 22/7 today.
 */
public void squareToRound(double diag, double diam) { ... }
```

Javadoc is written for classes, member variables, and member methods.

This is where the Java API documentation comes from!

In Eclipse: Project → Generate Javadoc

Java naming conventions

The Java Language Specification recommends these conventions

Generally: Use `camelCase` not `pothole_case`.

Class name: A noun phrase starting with a capital.

Method name: A verb phrase starting with lower case.

Instance variable: A noun phrase starting with lower case.

Local variable or parameter: ditto, but acronyms and abbreviations are more okay.

Constant: all uppercase, pothole.

E.g., MAX_ENROLMENT

Direct initialization of instance variables

You can initialize instance variables inside constructor(s).

An alternative: initialize in the same statement where they are declared.

Limitations:

- Can only refer to variables that have been initialized in previous lines.
- Can only use a single expression to compute the initial value.

What happens when

1. Allocate memory for the new object.
2. Initialize the instance variables to their default values:

   0 for ints, `false` for booleans, etc., and `null` for class types.

3. Call the appropriate constructor in the parent class.

   The one called on the first line, if the first line is `super(`*arguments*`)`, else the no-arg constructor.

4. Execute any direct initializations in the order in which they occur.
5. Execute the rest of the constructor.

Abstract classes and interfaces

A class may define methods without giving a body. In that case:

- Each of those methods must be declared `abstract`.
- The class must be declared `abstract` too.
- The class can't be instantiated.

A child class may implement some or all of the inherited abstract methods.

- If not all, it must be declared `abstract`.
- If all, it's not abstract and so can be instantiated.

If a class is completely abstract, we may choose instead to declare it to be an `interface`.

Interfaces

An interface is (usually) a class with no implementation.

- It has just the method signatures and return types.
- It guarantees capabilities.

Example: `java.util.List`

- "To be a List, here are the methods you must support."

A class can be declared to `implement` an interface.

- This means it defines a body for every method.
- A class can implement 0, 1 or many interfaces, but a class may extend only 0 or 1 classes.

An interface may extend another interface.

Generics: naming conventions

The Java Language Specification recommends these conventions for the names of type variables:

- very short, preferably a single character
- but evocative
- all uppercase to distinguish them from class and interface names

Specific suggestions:

Maps:	K, V
Exceptions:	X
Nothing particular:	T (or S, T, U or T1, T2, T3 for several)

Intro to Exceptions

What are exceptions?

Exceptions report exceptional conditions: unusual, strange, disturbing.

These conditions deserve exceptional treatment: not the usual go-to-the-next-step, plod-onwards approach.

Therefore, understanding exceptions requires thinking about a different model of program execution.

Exceptions in Java

To "throw an exception":

```
throw Throwable;
```

To "catch an exception" and deal with it:

```
try {

    statements

} catch (Throwable parameter) {  // The catch belongs
to the try.

    statements

}
```

To say you aren't going to deal with exceptions (or may throw your own):

Analogy

`throw`:

I'm in trouble, so I throw a rock through a window, with a message tied to it.

`try`:

Someone in the following block of code might throw rocks of various kinds. All you catchers line up ready for them.

`catch`:

If a rock of my kind comes by, I'll catch it and deal with it.

`throws`:

I’m warning you: if there’s trouble, I may throw a rock.

Examples

Dealing with exceptions:

- You will call many methods that can generate exceptions if something goes wrong. E.g., I/O methods.
- Example ...
- You can try-catch, or “pass it on”

Throwing your own exceptions:

- Example ...

Why use exceptions?

Less programmer time spent on handling errors

Cleaner program structure:

- isolates exceptional situations rather than sprinkling them through the code

Separation of concerns:

- Pay local attention to the algorithm being implemented and global attention to errors that are raised

(Bad) Example

```
int i = 0;
int sum = 0;
try {
  while (true) {
    sum += i++;
    if (i >= 10) {
      // we're done
      throw new Exception("i at limit");
    }
  }
} catch (Exception e) {
  System.out.println("sum to 10 = " + sum);
}
```

Why was that code bad?

The situation that the exception reports is not exceptional.

- It’s obvious that `i` will eventually be `10`. It's expected.
- Unlike Python, exceptions are reserved for exceptional situations.

It's uncharacteristic. Real uses of exceptions aren't local.

- `throw` and `catch` aren't generally in the same block of code.

We can have cascading catches

Much like an `if` with a series of `else if` clauses, a `try` can have a series of `catch` clauses.

After the last `catch` clause, you can have a clause:

```
finally { ... }
```

But `finally` is not like a last `else` on an `if` statement:
The `finally` clause is always executed, whether an exception was thrown or not, and whether or not the thrown exception was caught.

Example of a good use for this: close open files as a clean-up step.

An example of multiple catches

Suppose `ExSup` is the parent of `ExSubA` and `ExSubB`.

```
try { ... }
catch (ExSubA e) {
  // We do this if an ExSubA is thrown.
}
catch (ExSup e) {
  // We do this if any ExSup that's not an ExSubA is
thrown.
}
catch (ExSubB e) {
  // We never do this, even if an ExSubB is thrown.
}
finally {
  // We always do this, even if no exception is
```

Exceptions, Part 2

Recap

If you call code that may throw an exception, you have two choices.

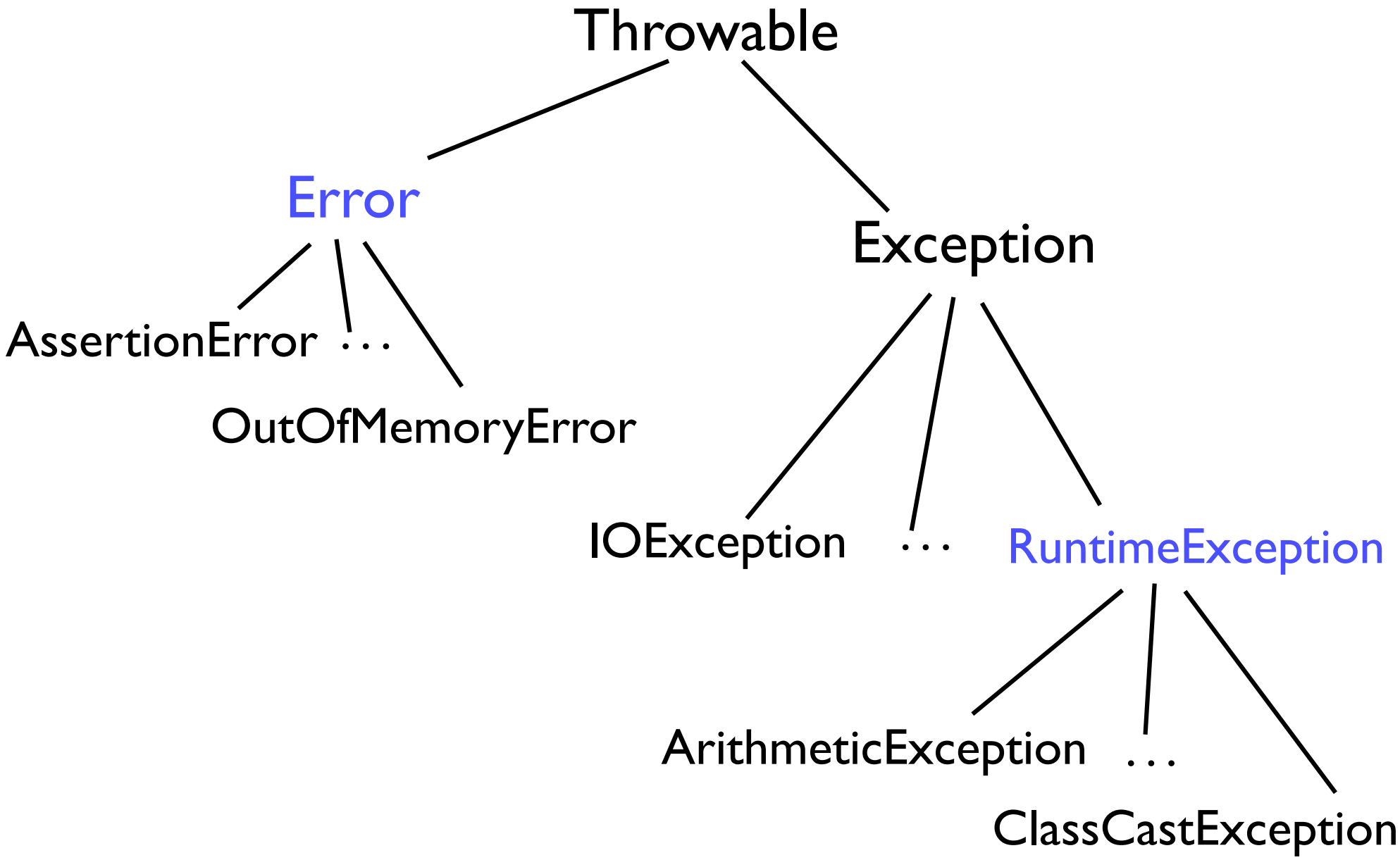
When you declare that a method “throws” something, you are reserving the right to do so, not guaranteeing that you will.

Exceptions don’t follow the normal control flow.

Some guidelines on using exceptions well:

- Use exceptions for exceptional circumstances.
- Throwing and catching should not be in the same method.
  “Throw low, catch high”.

Where `Exception` fits in



“Throwable” has useful methods

Constructors:

```
Throwable(), Throwable(String message)
```

Other useful methods:

```
getMessage()
printStackTrace()
getStackTrace()
```

You can also record (and look up) within a Throwable it’s “cause”: another Throwable that caused it to be thrown. Through this, you can record (and look up) a chain of exceptions.

You don't have to handle Errors or RuntimeExceptions

Error:

> "Indicates serious problems that a reasonable application should not try to catch."
>
> Do not have to handle these errors because they "are abnormal conditions that should never occur."

RuntimeException:

> These are called "unchecked" because you do not have to handle them.
>
> A good thing, because so many methods throw them it would be cumbersome to check them all.

Some things not to catch

Don’t catch `Error`: You can’t be expected to handle these.

Don’t catch `Throwable` or `Exception`: Catch something more specific.

(You can certainly do so when you’re experimenting with exceptions. Just don’t do it in real code without a good reason.)

What should you throw?

You can throw an instance of `Throwable` or any subclass of it
(whether an already defined subclass, or a subclass you define).

Don’t throw an instance of `Error` or any subclass of it: These are for unrecoverable circumstances.

Don’t throw an instance of `Exception`: Throw something more specific.

It’s okay to throw instances of:

- specific subclasses of `Exception` that are already defined,

Extending Exception: version 1

Example: a method `m()` that throws your own exception `MyException`,
a subclass of `Exception`:

```
class MyException extends Exception {...}

class MyClass {
  public void m() throws MyException { ...
    if (...) throw new MyException("oops!"); ...
  }
}
```

Extending Exception: version 2

Example: a method `m()` that throws your own exception `MyException`,
a subclass of `Exception`:

```
class MyClass {
  public static class MyException extends Exception {...}

  public void m() throws MyException { ...
    if (...) throw new MyException("oops!"); ...
  }
}
```

Aside: classes inside other classes

You can define a class inside another class. There are two kinds.

Static nested classes use the `static` keyword.

(It can only be used with classes that are nested.)

> Cannot access any other members of the enclosing class.

Inner classes do not use the `static` keyword.

> Can access all members of the enclosing class (even private ones).

Nested classes increase encapsulation. They make sense if you won't need to use the class outside its

Documenting Exceptions

```
/**
 * Return the mness of this object up to mlimit.
 * @param mlimit   The max mity to be checked.
 * @return int  The mness up to mlimit.
 * @throws MyException If the local alphabet has no m.
 */
public void m(int mlimit) throws MyException { ...
  if (...) throw new MyException ("oops!"); ...
}
```

You need both:

- the Javadoc comment is for human readers, and
- the `throws` is for the compiler.

Both the reader and the compiler are checking that caller and

Extend Exception or RuntimeException?

Recall that `RuntimeExceptions` are not checked. Example:

```
class MyClass {
  public static class MyException extends RuntimeException {...}
  public void m()  /* No "throws", yet it compiles! */  { ...
    if (...) throw new MyException("oops!"); ...
  }
}
```

How do you choose whether to extend `Exception` or `RuntimeException`?

Perhaps you should always extend `Exception` to benefit from the compiler's exception checking?

What does the Java API say?

`Exception`:

> "The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch."

`RuntimeException` (not checked):

> "RuntimeException is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine."
>
> `RuntimeException` examples:
>
> `ArithmeticException`, `IndexOutOfBoundsException`, `NoSuchElementException`, `NullPointerException`

`non-RuntimeException` (checked):

What does the Java Language Specification say?

"The runtime exception classes (RuntimeException and its subclasses) are exempted from compile-time checking because, in the judgment of the designers of the Java programming language, having to declare such exceptions would not aid significantly in establishing the correctness of programs. ... The information available to a compiler, and the level of analysis the compiler performs, are usually not sufficient to establish that such run-time exceptions cannot occur, even though this may be obvious to the programmer. Requiring such exception classes to be declared would simply be an irritation to programmers."

Reference:
http://docs.oracle.com/javase/specs/jls/se5.0/html/

Example

Imagine code that implements a circular linked list.

Suppose that, by construction, this structure can never involve null references.

The programmer can then be certain that a `NullPointerException` cannot occur.

But it would be difficult for a compiler to prove it.

> The theorem-proving technology that is needed to establish such global properties of data structures is developing, though. Wait 10 years. :-)
>
> Interested? Take CSC324 (programming languages) and CSC488 (compilers).

So if `NullPointerException` were checked, the programmer

Good advice from Joshua Bloch

"Use checked exceptions for conditions from which the caller can reasonably be expected to recover."

"Use run-time exceptions to indicate programming errors. The great majority of run-time exceptions indicate precondition violations."

I.e., if the programmer could have predicted the exception, don't make it checked.

Example: Suppose method `getItem(int i)` returns an item at a particular index in a collection and requires that i be in some valid range.

The programmer can check that before they call `o.getItem(x)`.

So sending an invalid index should not cause a checked