CS 211 Review Guide

Test #1 Review Guide

The first test covers everything we have discussed about basic Java programming, through control structures, arrays, methods, visibility, and inheritance (including abstract and final keywords when applied to classes and methods).

In preparing for your test, you should be aware that anything in the required readings, anything presented in lecture, and anything covered in the labs and assignments, is valid testing material. I will try to create a laundry list of topics here, but it is not a guarantee that things won't show up on the exam. Since I'm the one making the study guide and our test, it's a pretty good bet that it will indeed be the same listing of things I feel is important; just be sure you study everything at your disposal, even if this review guide serves to focus your time on specific topics.

This class, and especially this first test, has a significant focus on being able to write programs in the Java language. This course also has a significant focus on being able to describe the object-oriented concepts that we have learned, so as a transition from CS 112, there will likely be an increased (but not extreme) focus on more theoretical questions, such as "why is X a good idea", or "how does Y make Z exhibit more encapsulation?". We will still have questions that test your ability to program in Java, just as CS 112 would test your ability to write Python code. But now we will also have a slightly shifted focus towards the concepts of OO. Sometimes the theoretical part doesn't occupy quite as much a portion of our presentation time, because the details of writing Java code dwarf the theory; but that doesn't mean the theory was less important.

This guide does not attempt to go into complete detail, especially when that detail is available already in our slides and in the book.

Throughout the slides, there are many "Practice Problems" slides; they are an excellent starting point for the sorts of sample problems you might expect to see on the exam. Other examples (not explicitly listed as practice problems) in the slides are also quite instructive of what to expect on the test.

Java Basics

- The Java compiler translates from source code to bytecode, which runs on the "Java Virtual Machine". Many different systems have interpreters that translate bytecode to machine instructions for their particular CPU's.
- syntax: the way to write something in a language. semantics: the meaning of a language feature or specific piece of code.
- errors:
    - compile-time errors. E.g., syntax errors (can't be interpreted as valid code in the language); type errors (misuse of language features so that it is not valid code).
    - run-time errors. Code compiled, but attempted something illegal/impossible (e.g., divide by zero, out-of-bounds array index, using null like an actual object value).
    - semantic errors. The code compiles and runs (doing what it implied), but we have written code that doesn't do quite what we want to occur (such as dividing instead of multiplying, or not-quite-finding the maximum in an array).
- Edit-Compile-Run cycle: When writing code, we first edit a document; then we compile it (to translate to Java bytecode, in this class), which generates the .class file; last, we run it. No matter what development environment, or language, this three-phase cycle is occurring, even when the steps are blurred by DrJava or Eclipse or some other IDE.
- HelloWorld: be able to write out the "hello, world" program. Although we haven't learned what quite all the parts of it mean, we are making great progress already. Also, running any program amounts to this basic piece of code.
    - on the test, you won't always have to write code to make a full program; read carefully if a question just asks for a method, or lines of code that would do something when run. We're usually just looking for the couple lines or so that relate to a specific task.
- Whitespace: any whitespace allowed between any identifiers, operators, etc. (as long as they can be distinguished, we don't even need whitespace: x+=1*foo(a,b); versus x += 1 * foo (a , b) ;). Only exception: can't have newlines within a string literal (between matching ""'s). Thus whitespace doesn't have the indentation meaning that we saw in Python, even though we strongly encourage indentations again.
- Comments: /* multi line */ and //to-end-of-line styles.
- identifiers: names for things in our code. Consist of letters, numbers, underscores, and $'s, but don't start with numbers. (Please don't use $'s: they are for code that writes code).
- keywords: identifiers that have built-in meaning in the language, such as: for while if case switch public private …
- primitive types versus reference types:
    - primitive types: 8 basic types where the values are atomic (no sub-portions) and of a known, small, fixed size. They are: boolean, char, byte, short, int, long, float, double.
        - literals: know how to create literals in all primitive types. Recall hexadecimal input, F suffix for float, L suffix for longs.
    - reference types: types created through a class definition or array definition (whether we write the class or we get it from a library of code, such as java.util or java.lang).

- casting: a conversion from a value of one type to a value of another type.
    - implicit casting conversions: when no information is lost, Java will convert between types for us. E.g., from int to double; from short to int; from anything to String (for printing purposes, usually; uses the toString() method for objects).
    - explicit casting conversions: programmer-specified conversions. Usually required when the conversion loses precision or other information; tells Java that it is 'okay' to perform the lossy action. E.g.: from double to int; from long to short.
    - Be able to identify when Java performs implicit casts.
    - Be able to use explicit casts to get data into the right format (type).
- Creating Variables
    - declaration: a type and an identifier. Tells Java that there should be a named storage location that can contain one value of the given type, and will be accessed using the given identifier. Only declared variables can be used, ever.
- Expressions vs. Statements
    - Expression: a representation of a calculation that can be evaluated to result in a single value; no indication what to do with the value.
        - Be comfortable identifying expressions.
        - Understand the ternary ?: expression.
    - Statement: a command/instruction for the computer to perform some action; often, statements contain expressions.
        - Be comfortable identifying statements versus expressions.
- Strings
    - concatenation: + operator.
    - implicit conversion of everything to String values when added to a String, or when a String value is needed.
    - escape sequences: \t, \n, \", \\, etc.
    - checking for string equality: can't just use ==. Use the equals method: e.g., s1.equals(s2).
    - Strings are immutable (can't change).
- printing: via System.out.println(), System.out.print(), System.out.printf().
- Scanner: class providing convenient methods to read text input from a source (such as keyboard/System.in).
    - Be able to use basic scanner methods: nextInt(), nextDouble(), next(), nextLine(), etc.
- Constant: a 'variable' whose value will never change from its initial value. Indicated with the **final** keyword.
    - naming convention: all caps. ex: MIN_HEIGHT, NUM_PONIES.
- increment/decrement: e.g., x++, ++x, x--, --x. "syntactic sugar" for an increment on the previous or following line. See our slide examples for their semantics. Be ready to examine code using them and report what happens. We ought not have more than one of these per variable per statement…

Control Flow

- heavy use of boolean expressions. (expressions that result in a boolean value).
- if/if-else. (No elif: since we use {}'s instead of indentation, we can get the same effect with chained if-else's).
- switch: old-fashioned, fast way to branch.
    - can only switch based on primitive value (or enum, or String as of Java 1.7).
    - cases must be **values**, not expressions.
    - use break statements to manually leave after each case statement (or don't, with unusual control flow behavior compared to if-elses).
    - default case can catch all non-cased values.
- while loop: keep executing body as long as guard (boolean expression) is true. Body might execute zero times (if guard is false first time).
- do-while loop: like while loop, but guaranteed to run at least once (guard checked **after** each loop iteration, not before).
- for loop: init, guard, update. See slides for detailed examples/while loop version.
- for-each loop: requires an 'Iterator' (arrays are Iterators). Looks like value-based for loop from Python; avoids index usage but still lets us access/modify each element in the Iterator (array, for us for now).
- break, continue: ways to modify loop/switch control flow. Try not to use them too often, but sometimes they really do make a block of code simpler.

Arrays

- arrays are fixed-length sequences of values, all of a single type. e.g., an array of integers (where each slot holds one integer, like an integer variable).
- representing an array type: place []'s after a type to indicate an array of that type. String → String[]. int → int[]. Defines an array type.
    - can add multiple dimensions: int[][], float[][][], etc.
- Declaration: as always, give a type and an identifier; now the type is an array type. ex: int[ ] xs ;
- Instantiation: must indicate exact length.
    - {vals,like,this} //allowed at declaration time only.
    - new int[]{vals,like,this} // allowed anywhere
    - new int[10]. (Fills with default values: 0, false, null, as appropriate for the type).
- accessing/update: using xs[index]. index must be an expression yielding an integer value.
    - No slicing like Python allowed.
- usage with loops. A lot of code examples used loops and arrays; be comfortable doing these sorts of things.

Classes and Objects

- Class: a class is a type. It is like a blueprint for making objects. It defines the state that each object will have, and what behaviors (methods) are available for those objects.
- Object: an object is a value. (A value of a particular class type). It is an *instance* of its class (one distinct value of that type). It has its own copy of each instance variable and method.
- Terminology: a type defined through a class is called a "reference type", because we don't have the direct value as with ints; instead, we deal with references to the object value that resides in memory.

Variable versus Reference versus Object:

- A variable is a named container (on the stack). A reference is just a (type,address) pair of an address of an object in memory that also knows what type it points to. Some variables only contain primitive values (no reference), while other variables only contain references (the value is elsewhere). An object is an instance of a class (it's a value), and it resides in the heap memory. The object contains its own copy of each instance variable and method.
- many references can have the address of the same object; they are called aliases. Since there's only one object involved, updates via one reference are visible through the other references.
- instance variable: a (non-static) variable declared inside a class, indicating that each object of the class will have its own maintained copy of this variable. E.g., the Square class has a side instance variable. A Coordinate class could have both x and y instance variables.
- constructor: special method that is used when creating a new object.
    - no return type listed (returns reference to object of the class's type)
    - method name must be the class name exactly.
    - parameters: entirely at programmer's discretion. (Often one per instance variable).
    - default constructor: If a class definition does not explicitly list a constructor definition, then a default implementation is available: no parameters, and all instance variables receive default values: 0, false, and null (for reference types).

Methods

- named block of code that can be called.
- defined in a class → thus has access to things defined in the class (no matter what visibility).
- available modifiers: visibility (public/private/…), static or not, final or not, others we haven't seen.
- method signature: modifiers, return type, name, parameter list.
    - example: public static void main (String[] args)
- parameters: formal parameters defined in parameter list (declares them); when a method is called, the actual parameters (arguments) are supplied to instantiate the formal parameters for this particular invocation of the method. Parameters are local variables.
- return type: Java enforces that the method will always return a value of the specified return type.
    - void: indicates that control flow should return without a return-value. Java enforces this, too. All possible paths must guarantee a return of the correct type of value.
- method overloading: when two methods share the same name, but are still distinguished by their parameter lists, Java allows them to coexist (never ambiguity which one is being called).
    - only the name, number of params, and types of params can distinguish them.
    - constructor methods can also be overloaded! This is a great thing.
- Control Flow: understand how control flow passes through multiple methods as they call each other. Our diagrams of the stack of method frames exhibited both this control flow as well as the location of local data (in the frames, dying as methods were exited) and objects (in the heap – meaning space to the right in our diagrams, which could last arbitrarily long, as long as our references meant Java wouldn't garbage collect them).

References

- the result of a constructor call is not the object itself – it is a reference to that object.
- A reference is simply an address of an object.

Method Calling Conventions (how are actual parameters actually transmitted?)

- only values are ever sent across from actual parameters of a method call to formal parameters during execution of method body.
- primitive types: a copy of the primitive value is sent. No effect on the original primitive value is possible (or wherever it may have been stored).
- reference types: a copy of the reference value is sent (creating an alias). No effect on the original *reference* is possible; however, these aliases both point to the same object in memory and thus can witness each others' updates to that object.
- We had examples of each of these, to understand what effect on memory was possible when passing primitive/reference types for parameters.

Static

- modifier that indicates there should only be one definition for the entire class.
- Note: static doesn't mean "unchanging"! That's what the final keyword is for.
- static variable: a "class variable". One copy overall, regardless how many objects are made (zero to many, it doesn't matter). Access as `Classname.varname`, or, as `varname` inside the class.
- static method: like a regular method, but cannot use instance variables. Therefore, accessible without an object, directly through class (since no object is used to call it). Access as `Classname.method(..)`, or inside the class definition as just `method(..)`
- Usage example: main is static (because we don't make an object of the class to run main). If we want 'helper' methods for main, they must also be static:
    - static things can't use non-static things! (Local variables in a static method seem like an exception, but it's still inside a static location).

Object class

- ancestor of all classes. Provides the toString() and equals(Object o) methods, among others.
    - we can redefine toString() to aid in printing (Java uses this toString method to String-ify any object, quite often with printing in mind).

Scope

- scope of data is the area in a program where that data can be referenced (used).
- data declared at the class level can be referenced by all methods in that class.
- data declared within a method is called "local data", and can only be used in that method.

Terminology

- variables defined in a class, whether static or non-static, are called fields.
- fields and methods are all called "members" of a class.

Visibility

- class members (and classes) can be given a visibility. For now, just consider public and private.
- public: anyone with access to this object is allowed to use this public portion of it (whether it's reading/writing a public variable, or calling a public method).
- private: access to this member is restricted to other members inside the class: meaning that an object can use this private thing while performing its own calculations, but the outside world can't use it.

    - Good for enforcing encapsulation: presentation to outside world is the public stuff, internal-only representations and methods are private stuff.
- Based on current view of an object: internal vs. external. (internal: everything available; external: only public stuff available).
- public methods: "service methods". private methods: "support methods".
- accessor/mutator methods (getters and setters): ways to individually restore read/write privileges to users of private variables.
- public stuff defines the "interface" (we called it the API/application programming interface in Python) to the object.

Inheritance

- Allows a new sort of code reuse: similar state (fields) and behavior (methods) can be "inherited" from one class to another.
    - Establishes a "parent-child" relationship. Also called "superclass/subclass", "base class/derived class".
    - indicated with `extends` keyword: `class Car extends Vehicle`
- Perhaps most importantly, we now have a supertype/subtype relationship. All the child classes' objects can be used where the parent class's objects were expected.
- Inheriting things: any fields or methods of the parent class are automatically a part of the child class. They can never be removed. Methods may be re-implemented (with the parent class's approval – non-`final` methods).

- **Visibility:**
    - public: anyone that can name it can access it.
    - protected: anyone in the package, and child classes outside the package, can access it.
    - default (package-private vis.): all in package can see it; nobody outside of the package, not even children, can see it.
    - private: only visible to this specific class (not to children or co-package members).

- `super`: refers to the parent class.
    - note: `this` referred to an *instance*; super refers to a *class*.
    - uses: calling parent constructors: `super(any,args)`
      finding shadowed (overridden) parent members: `super.fieldName`

- **constructors:** only thing that isn't directly inherited (we *must* make our own constructors too).
    - a child class constructor *must* call its parent constructor as its very first statement, using `super(any,args);` and correctly matching the parameters list of an actual constructor for the parent class.
        - Java actually adds `super();` to any child class constructor that doesn't explicitly call its parent constructor. If no such zero-parameters constructor existed in the parent class, it is a compilation error.

- **single inheritance:** Java allows exactly one parent class, always.
    - if no parent class is specified, the `Object` class is the parent class.
    - we can simulate multiple inheritance with *interfaces* later on.

- **Overriding Methods:**
  - child class can provide a new implementation of a method inherited from parent class.
  - the method signature must *exactly* match: name, parameters' types / ordering / number, and also the return type.
  - No matter at what type we're viewing the child object, if we call the overridden method, we get the child's specialized implementation.
    - we call methods on objects; whenever that particular object was instantiated it had a specific type (the class containing that specific constructor used)
      → That type *always* dictates what versions are used, no matter where else (and at what type) we eventually call methods on that object. e.g., we'd get the same implementation of `makeNoise()`:

```
Labrador spot = new Labrador(..);
spot.makeNoise(); // uses Labrador version.
((Dog)d).makeNoise();   // still uses Labrador version.
```

- **Overriding Fields** (a bad, bad idea):
  you can also override fields, but my opinion is that this will almost always be a programming bug – competing versions of the same-named variable (perhaps different types), where inherited methods use the parent's version, and new/overridden methods use the child's version. Ugly bugly!

- **Difference: Overloading vs. Overriding**
  - overloading: providing same-named methods that purposefully have different method signatures (parameter list #/types), to provide multiple implementations based on different inputs. Especially useful for constructors. Possible without any inheritance involved. ***The different versions co-exist in peace.***
  - overriding: requires parent/child, inheritance. Child class inherits method, yet re-defines what it will do when that method is called by purposefully having the *same* method signature, but providing a new body. ***There can be only one!***

- **Object class**
  - it is always an ancestor (parent, grandparent, ..) of every class.
  - provides a few useful methods: `equals(..)`, `toString()`, others.
  - we prefer always overriding these methods.

- **Abstract class (PUSHED BACK TO TEST #2):**
  - a class that cannot (yet) be instantiated (we can't make objects of this exact type).
  - The opposite of an "abstract" class is a "concrete" class.
    - concrete classes may be instantiated
  - abstract class: achieved by adding the `abstract` keyword to class declaration.

```
public abstract class Shape { .. }
```

  - abstract classes also may optionally include abstract methods: methods that are declared (with abstract modifier), but have no body (a `;` instead of `{..}`).
    - just like all other members, abstract methods will be inherited.
  - abstract classes may still have child classes. If the child class overrides every abstract method that was inherited, then the child may be concrete.
  - Just like other classes, an abstract class is a type. The collection of fields and methods we introduced are all guaranteed to exist (and be concrete) for all actual instances from child

classes. This is the whole point of introducing an abstract class into our class hierarchy: to have a formal way to group related child classes and use them uniformly.

- **`final` classes, `final` methods:**
  - if a class is declared to be `final`, it may not be extended (no child classes are allowed).
  - if a method is declared to be `final`, it may not be overridden (no child class may ever change its implementation of this method).

Test #2 Review Guide

Remember, the abstract keyword (methods, classes) is also on this test.

In preparing for your test, you should be aware that anything in the required readings, anything presented in lecture, and anything covered in the labs and assignments, is valid testing material. I will try to create a laundry list of topics here, but omission here is not a guarantee that things won't show up on the exam. Since I'm the one making the study guide and the test (for Snyder's sections, anyways), it's a pretty good bet that it will indeed be the same listing of things I feel is important; just be sure you study everything at your disposal, even if this review guide serves to focus your time on specific topics. This guide does not attempt to go into great detail (such as exhaustive examples), especially when that detail is available already in our slides, lab tutorials, and in the book.

As is often the case, if you are not running Java code while you study, you will likely not do well on the test. Don't restrict yourself to just catching up on reading; be sure you actually practice writing code! Throughout the slides, there are those "Practice Problems" slides; they are an excellent starting point for the sorts of sample problems you might expect to see on the exam. The quizzes are another excellent resource for sample question styles. Other examples (not explicitly listed as practice problems) in the slides are also quite instructive of what to expect on the test. We've posted a lot of code samples from class; review that code as a way to see the ways we introduced topics, and think back to how we discussed why we would do things, followed by how we would do them. Lastly, if you haven't completed the lab tutorials yet, that's a great source of practice! Don't just skim over examples because you don't see any difficult situations arising; code it up, make sure it compiles, make sure it works. I have a hunch people are omitting the lab chapter exercises, and that might be a telling factor in who does well on the test or not.

Format will again likely be part multiple choice, part short answer, part longer responses.

Topics

Packages

- means of grouping Java classes for distribution/usage in multiple projects.
- utilizes actual folder structure to make package structure. Files must include package statements.
- aids code reuse, avoids name clashes, organizes code.
- some class libraries (packages) are provided with Java: java.util, java.lang, etc.
- a jar file is a "Java Archive": a single-file approach to providing entire libraries of Java classes (still maintains package hierarchy). It is actually just a zip file!
- Classpath
    - specifies where Java (javac, java) should look for class definitions. By putting a package's directory on the classpath (or a jar file directly on the classpath), Java can find all classes defined in the package.
- package usage:
    - we could just fully qualify a class from a package at each and every usage:
        - java.util.Scanner sc = new java.util.Scanner(System.in);
    - But we tend to import it first:
        - import java.util.*; // outside of class
        - Scanner sc = new Scanner (System.in); // inside method
    - Consider our package example.

Interfaces

- Java's "controlled simulation of multiple inheritance".
- An interface is a type.
- defines a group of *abstract* methods that *any* class may implement.
    - an interface *never* has any fields. Add getters/setters to pretend.
- A class "signs the contract" of an interface by claiming it implements it:
  ```
  public class Car implements Sellable { ..
  ```
- A class "fulfills the contract" by actually implementing (overriding) every single method of the interface. If even a single method from the interface is left out, the class must be abstract (allowing for abstract methods to be passed on to child classes), or else it didn't actually implement the interface, and a compiler error occurs.
- Examples of interfaces in Java's standard libraries:
    - `Comparable`: provides just the `compareTo(..)` method. Allows an understanding of ordering. Useful for sorting and such.
    - `Iterator`: provides `hasNext()`, `next()`,and `remove()`.
    - `Serializable`: no methods. But clues Java in to make the class representable for storing in a file or transmitting over a network.
- Advanced interfaces:
    - interfaces may inherit from each other! We didn't try this, but it's interesting how rich and related the types we can introduce in Java may be.

Enumerations

- basically, a finite set of values that are grouped into a new type.
- simple examples:
    - `enum TrafficLight {RED, AMBER, GREEN}`
    - `enum Day {MON, TUES, WED, THURS, FRI, SAT, SUN}`
- internal implementation (good mental model): as a class, except that constructors must be private (so no extra values are created anywhere else), and the finite list of values are included at the beginning of the class definition once and for all (as static fields). Otherwise, we can still add methods, fields, all the usual class stuff.
- Simple Usage:
    - in general, access with `EnumName.EnumValue` (e.g., `Grade.A`).
    - in switch: access with `EnumValue`. (Java already knows `EnumName` b/c of the switch expression)
    - in foreach loop (utilizing the array returned by values()):
      ```
      for (Grade g : Grade.values() ) { ... }
      ```
- Advanced features:
    - add three things at once:
        - fields (any visibility)
        - private/package-default constructors (not public: can't be accessed outside package ever, and no protected: enums can't be extended)
        - "constructor calls" to the enumerated values to give state to the values (see slides)
    - add methods to the entire enumeration. (static or non-static; different visibilities ok except protected)

Exceptions

- Idea: Java's representation of "something went wrong that can't be handled right here".
  - when thrown, an exception causes normal control flow to be abandoned, like a rogue return statement.
  - There are many Exception classes, related in a class hierarchy (using inheritance). We can extend these classes with our own child classes, letting us hook into Java's exception and exception handling effort.
  - Exception classes to know about (know of an example if starred):
    - `Throwable`
      - `Exception`
        - `RuntimeException`
          - `NullPointerException`
          - `ArrayIndexOutOfBoundsException`
          - `ClassCastException`
          - `ArithmeticException`
        - `IOException`
          - `FileNotFoundException`

- **Causing exceptions:**
  - call a method that throws an exception (e.g. , file not found)
  - expression that's nonsensical (e.g., `5/0` or `xs[-3]`)
  - manually throw one (e.g., `throw new MyException("☹");`)
- **Handling exceptions:**
  - wrap the suspicious code in a `try` block.
  - immediately-following `catch` blocks specify how to handle exceptions of the particular type listed in the `catch` block.
  - we can have multiple catch blocks for one try block, where each catch block handles a different exception.
    - order catch blocks from most specific (most childly) to most general (most ancestral, such as `Exception` itself).
  - we also may have a `finally` block after all the catch blocks. This block always runs – whether no exceptions occur, or an exception occurs and is caught, or an exception occurs and won't be caught (the finally block still gets to run before the propagation continues, and if the finally block throws its own exception then the original one is lost).

- **Propagating exceptions:**
  - means we allow it to crash this part of the program, if the exception actually occurs.
  - when a `try` block has no corresponding `catch` block (or we didn't even have a `try`).
  - ***Checked*** vs. ***Unchecked*** exceptions:
    - checked exceptions that are propagated must be admitted in the signature:
      ```
      public int foo (int a) throws IOException {..}
      ```
    - unchecked exceptions that are propagated don't have to be admitted, but may be.
    - unchecked exceptions are the **`RuntimeException`** class, **`Error`** class, and their children. They tend to represent program bugs and hardware failure respectively, so should be solved/fixed (for bugs) or dealt with gracefully.
    - checked exceptions represent things beyond programmer's control that still must be dealt with. e.g.:
      - `FileNotFoundException` when trying to read a file
      - `InputMismatchException` when Scanner token is wrong

- side effects and control flow:
  - any side effects (reassigning a variable, printing) that occurred before the exception cannot be undone.
  - the current statement isn't even finished (so maybe a method call wasn't attempted if the argument's evaluation generated an exception; or, the assignment statement didn't happen because we generated an exception trying to get a value for it).
  - blocks of code are abruptly exited in search of an enclosing `try-catch`-block, meaning some side-effectful lines that would normally have been executed next will instead not be run.

- **Creating your own exception classes**
  - write a child class of any throwable class (includes all the Exception classes mentioned).
    - Extending `RuntimeException` (or `Error`) makes yours unchecked; extending `Exception` makes yours checked.
    - Any (non-final) Exception class may be extended.
    - Use of fields, methods, overriding toString() highly encouraged.
  - Create objects of your exception class as normal (with constructor call):
    - `MyExc me = new MyExc ("problem studying", 410);`
  - throw your exception manually:

```
throw me; //following previous code snippet
```

Wrapper Classes

- class-versions of the primitive types.
  - Sometimes only objects are allowed; these also can group more info, like methods and static things, with the type.
- Auto-boxing: casting between the primitive types and their wrapper-class equivalents occurs implicitly (automatically). Java is able to meaningfully convert between them so it occurs.

Command-Line Arguments

- Since running a Java program means running the main method, we can pass in an array of String arguments on the command-line when we execute our Java program:

```
demo$ java MyProgram args here 123 "spacey ones in quotes" too
```

Only strings available: main method signature has `(String[] args)` parameter list.

- values are separated by spaces. matching single- or double-quotes may be used to create a single String that contains a space (e.g., `"spacey arg"`).
- getting numbers: use `Integer.parseInt(s)` or `Double.parseDouble(s)`, based on any `String s`. (e.g., `Integer.parseInt(args[0])`).

Testing and Tools

- **javadoc:** a tool for generating API documentation based on Java packages (and classes).
  - generates a lot of HTML (good to put in separate directory. Then, open index.html)
  - always "scrapes" source code for things like inheritance hierarchy, implemented interfaces, listing out fields/methods.
  - can write special /** javadoc */ comments that are also inspected by javadoc.
    - use @tags to label more info: @param, @return, @exception, @author, etc.
  - can limit what levels of visibility are included
  - can generate API documentation for many packages/sub-packages all together.
  - might add package-info.java files to document entire packages.
- **debugger:** a tool for interactively inspecting values of a running program.
  - breakpoints: when these statements are reached, execution pauses (allows user to take control and inspect/change things).
  - stepping: advancing one statement at a time.
    - step into: allow steps to travel with method calls and step there too. (follow calls)
    - step over: allow steps of method calls to call-return in one atomic step. (stay local)
    - step out: keep stepping until a return statement is reached (let's leave now)
  - watches / watch lists: a set of variable names that the debugger will show values for at each pause (of course, can't display anything when nothing of that name is in scope). Just variables/fields available. *(DrJava now allows tracking of expressions, cool!)*
  - modifications:
    - user can directly change a variable's memory to see effects (but doesn't change what original program will do when run again)
    - user can run expressions/statements interactively (to inspect more than just variables).
    - direct edits to source code generally will **not** affect debugger's behavior until next compilation/debug-run.
- **Testing**
  - gaining assurance that code works as expected
  - catching bugs in earlier phases saves orders of magnitude of effort!
  - test case: specific inputs and expected outputs for some part of the program.
  - regression testing: after making any changes, we re-run old tests that had passed, to double check that nothing was broken.
  - Testing styles:
    - **black box testing:** test cases that focus on the meaning of the code: with no view inside the "black box", what behavior is expected?
    - **white box testing:** test cases that are aware of the particular implementation, and attempt to get *code coverage*. Each part of the code (loop, if-else, etc) gets test cases to try and use that piece of code. In general, test cases that focus on the implementation more than the general meaning of the code could be considered white box testing.
  - passing all tests doesn't mean program is necessarily correct. (Can't usually test all cases anyways). Defining correctness is very hard!
  - unit testing: writing tests for the smallest parts of a program.
  - test-driven development: writing tests before (or at least alongside) implementation

- **JUnit:** a unit testing framework for Java.
  - java package that can be imported (and used to run test cases/report on results)
  - offers annotations: @Test methods are test cases. Also, @Before, @After.
  - require behavior by using assertWhatever methods: `assertEqual(..)`, `assertTrue(..)`,`assertNull(..)`. Report failures with `fail()`.

Generics

- generics allow us to add type parameters to classes or methods (or interfaces).
  - This gives us parametric polymorphism – entire class / method / interface definitions parameterized over **any** type.
  - type parameters on a class let us effectively create a whole batch of class definitions: `public class ArrayList<E> { .. }` gives us `ArrayList<String>`, ArrayList<Integer>, `ArrayList<Person>`, and `ArrayLists` of any other type. Even more exotic ones, like `ArrayList<ArrayList<Integer>>`.
- Problem that is solved: Java sometimes "loses" a type. The original `ArrayList` only stored `Object` values. Even if we only put `Integer` values in it, Java could only remember that it was an `Object` value, requiring lots of casting. (For instance, inside foreach loops). The programmer is now responsible for this "type checking", and can easily mess up.
  - Generics allow use to introduce a type parameter that, once instantiated, gives Java a reminder of the only acceptable type of inputs that are allowed for this particular `ArrayList` value.

- Class-definition generics
  - we may introduce a list of type parameters in a class declaration.
    - example: `public class Pair <R,S> { R first; S second; }`
  - those types are now usable anywhere inside the class definition (e.g., for field types, return types, parameter types, and local definition types). Note: no extra < >'s here after the type parameter list
  - the type parameters must be instantiated at each constructor call, locking in the correct type

- Method-definition generics
  - we may introduce a list of type parameters in a method declaration.
    - example:

```
public <U> U choose (U left, U right, boolean selector) {
     return  b ? first : second;
}
```

    - Not connected to class type parameters: these are only instantiated/used inside the method, and possibly at different types for each call of the method.
    - Offers a sort of "infinite overloading", but lets us constrain different types via equality of the same type parameter, though it will become some specific type whenever the method is called.
- Our Examples:
  - we looked at a simplistic view of the ArrayList class as an example of generics. We also had extensive examples of Pairs and Boxes in the lab.

- Going further:
  - There are more advanced notions of generics, involving notions of subtyping, extending multiple types (both from class inheritance and interfaces, despite frugal re-usage of the extends keyword), and even wildcards. Just be aware that we can get more than this basic parametric polymorphism.
- Example usage: ArrayList.
  - Supply a type in <>'s to create the actual type, which then is used to declare variables or call constructors: (all in one line here:)

```
ArrayList<Integer> intlist = new ArrayList<Integer>();      // decl/constr: give type params.
intlist.add(5);   intlist.add(7);    intlist.add(12);       // usage (no explicit types are given)
int sum = 0;
for ( Integer i : intlist) {   sum += i;  }                 // Java knows only Integers are present.
```

Java Collections

- Java standard library versions of common patterns of data.
- Implemented through a series of inheritance-linked interfaces, with many generics-enhanced classes implementing these interfaces through different means (e.g., ArrayList<E> class and LinkedList<E> class both implement the List<E> interface with different underlying data representations and correspondingly different behaviors).
- Shining examples of data structures! (A taste of things to come in CS 310).

ArrayList<E>

- part of the Java Collections Framework, ArrayList<E> represents a List<E> structure (and has an underlying array implementation, hence the name "ArrayList").
- As a class, all list operations are through methods: e.g., `xs.get(i)`, `xs.set(i,val)` instead of `xs[i]`.
- as a list, it has more functionality than just an array: we can add/remove elements (changing the length of the structure), and we can do so at any index (not just the end). Feels like Python lists (because it is a list, not an array!). All these behaviors come from implementing List<E>.
- extra methods available due to the array implementation: ensureCapacity, trimToSize, more constrs.
- Uses generics to allow it to contain any specific type of element. (see generics for more info).

CS 211

Final Review Guide

This review guide covers the last third of the course (everything after our second test's materials). Given the timing of our tests, this is really more like a quarter of the semester.

The final exam itself will be cumulative: you should anticipate roughly 1/3 the exam to cover the materials listed here, and the rest should cover the materials of the first two tests. The lab manual chapters go into far better details than will be shown here; you should consider the lab manual readings, lab exercises, quizzes, tasks, and slides as other good sources for ideas on what is important, as well as what sorts of questions might be asked. Of course, the textbook readings should have been completed by now as well. Good luck!

Recursion

- Recursion generally means that something is defined in terms of itself.
  - code can be recursive – a method may call itself.
  - data can be recursive – a class may contain a field whose type is the class type itself.

- Code Recursion
  - Split the possible actions into base cases and recursive cases.
    - Base Case: for specific inputs, we memorize/know the answer, and report it directly without having to call ourselves.
    - Recursive Case: for a given input, we know how to rephrase the solution in terms of a call to the same method on "smaller" inputs (closer to a base case).
  - Recursion(through recursive cases) must be terminated (by reaching a base case), or else it's just like an infinite loop: never ending. (Actual result would be a `StackOverflowError`).
    - The recursive calls of a method must be on a "smaller" version of the problem. This could mean subtracting one from some counter, calling on a list whose length is shorter than the current call's arguments, or anything that represents progress towards a base case.
  - Each recursive call is distinct: it has its own frame on the stack, meaning it has its own copy of local data (parameters, local declarations), and after the recursive call it made is complete, it will continue its own execution until returning.
  - Base cases must be tested before recursive cases, to ensure the recursion can stop.
    - error conditions (e.g., negative numbers for factorial) can be handled like base cases.
  - Recursion can be used to solve the same problems that iteration can – just as we can choose between a for-loop and a while-loop, we can choose between recursion and iteration. Each one is better-suited to certain problems.
    - Often, definitions themselves are recursive, so it's convenient (translation: low cognitive effort) to write the definitions the same way.
    - Iterative definitions are often faster by a negligible to noticeable amount, so it might be worth the effort to craft an iterative implementation.

- Recursion is sometimes far less efficient than an iterative approach. For example, Fibonacci can lead to a drastic amount of duplicated calculations. This is certainly not automatically the case for recursive solutions, but it means we must be conscious of the decision to write a recursive or iterative solution. To spot these cases, consider the size of your input (e.g. length of array, value of some number) and think about if the number of recursive calls could be as large as the input size (recursion might be okay), could be far larger (recursion is probably a bad choice), or could never be more than a fraction of the size (like log or gcd, where recursion is probably a very nice option). If the number of recursive calls is based on one variable's value like in Fibonacci, then that is a horrendous time to use recursion.

- Data Recursion
  - Data can be recursive: the definition of the data's structure may include a field of the type being defined.
  - The base case is null.
  - The recursive case is the field of the same type.
  - Data Structures are often written with recursive definitions. Now the notion of some value may be represented by many objects of the recursive type, instead of one. This is really the same as an object having fields, except that the fields happen to be of the same type in this recursive example.
  - Example: Linked Lists can be represented as two fields: a value and another LinkedList. (see the lab for more details)

Searching and Sorting

- Searching: given some collection of data, and some key that will identify the piece of data we seek, search through that collection and return the data.
  - some possible returns: a reference to the object; the index within the collection that can be used to quickly re-find the data.
- Sorting: given some collection of data, and the notion of some sequential traversal, relocate values within the collection so that the traversal will visit values in some ordered fashion.
  - examples: sorting an array of Person objects by name, or alternatively sorting by age. Sorting numbers ascending.
- You should be capable of writing a search algorithm and of writing a sort algorithm of your choosing. (Easy choices would be linear search and bubble sort).

Searching

- Basic Searching Example: Linear Search
  - assuming no ordering of the data, must check every location for the desired value.
  - just loop over each location and check for a match.
  - for a list of length n, might take n inspections before completing (complexity: O(N)).
- Faster Searching Example: Binary Search
  - assumes/requires that the data are sorted in order.
  - given a range that would contain the key if present, check the middle value. Either it's the match (done), or else we know that the key would have to be in just the lower half, or just the upper half, of our range. Search again in that smaller half.
  - repeatedly search on that side range, until either a match is found, or there is no place left to look (the remaining range is of size zero).
  - for a list of length n, only requires $\log_2(n)$ recursive calls at worst to find the key. (complexity: O(lg(N)).

Sorting

- Basic Sorting Example: Bubble Sort
  - idea: one traversal will step through the list from front to back, comparing each adjacent pair of values, swapping if they're out of order.
    - each traversal guarantees that at least one more item is sorted at the end – larger values 'bubble' up to their correct location.
    - after n traversals (for a list of length n), the entire list is definitely sorted.
    - complexity: $O(N^2)$.
  - improvements to the basic double-for-loop algorithm:
    - each successive iteration doesn't have to run all the way to the end of the list – after k traversals, the last k elements don't need to be compared.
    - if no swaps occurred in a traversal, the entire list is sorted – stop!
- Selection Sort:
  - idea: keep finding and swapping the smallest remaining value to the earliest unsorted spot.
    - find the minimum value in the list (traverse it)
    - swap this min-value to the lowest unsorted spot in the list. That least spot is done.
    - keep repeating min-find and swap to lowest unsorted spot until sorted portion grows to cover entire list.
  - complexity: still $O(N^2)$.
- More-Advanced Sorting Example: Merge Sort
  - idea: we realize that merging two already-sorted lists into a single sorted list isn't very hard. So keep splitting the list in half, until we have lists of length 1 (or 0), which are already sorted. Then re-merge them together preserving the orderedness. Easily defined via recursion, though not necessary for an implementation.
  - merging two lists together: keep inspecting the front item of our two lists; the lower value is removed from that list and added to our result.
  - the simple approach of creating two sub-lists: copies all the values, resulting in a lot of space usage.

Search and Sort Summary

- There are many other ways to search or sort data, we simply looked at a simplistic and then more involved version of each.
- Different issues of efficiency, both in time taken and space needed during the calculation, arise.

Compara(tor/ble)<T>

- both provide ways to compare two objects of the same type. The less-than, equal-to, and greater-than relations are indicated with negative, zero, or positive return values respectively.
- `interface Comparable<T>`
  - when there's one best way to compare, class Foo should implement Comparable<Foo>.
  - `public int compareTo(T other)`
- `interface Comparator<T>`
  - Should be used when either no Comparable<T> instance was provided, or we want a different meaning of comparison (e.g. comparing people by name instead of age).
  - This "bystander" object will look at two T values, and return the relation as an int as above.
  - `public int compare(T arg1, T arg2)`

Extra Topics

We covered many extra things at the semester's end; though the test won't ask you to read or write code on the following, they are both useful as a programmer, and for possible extra credit questions.

Anonymous Classes

When we need a single object of some class – whether to make a concrete object from an abstract class, to provide an object that implements an interface, even odder situations like filling in abstract methods in enumerations – going through the effort of creating a separate class and invoking its constructor just once is cumbersome. Anonymous classes let us create this one-use class and the object of it. We supply what looks like a constructor call, followed by {}'s containing any extra needed definitions.

```
abstract class AlmostComplete { int c;    AlmostComplete(int c){this.c=c;}    public abstract void increment(); } }
interface Predicate<T> { public boolean test(T t); }
enum Mulls {                                  // anonymous classes to provide int go(int) implementations
  twoTimes  (){@Override int go(int x){ return x*2;}},
  threeTimes(){@Override int go(int x){ return x*3;}},
  fourTimes (){@Override int go(int x){ return x*4;}};
  abstract int go(int x);
}
...
AlmostComplete ac = new AlmostComplete(5){ @Override public void increment() { c++ ; } };
Predicate<Integer> over5 = new Predicate<Integer>(){ @Override public boolean test(Integer i) { return i>5; }};
ac.increment();
System.out.println(over5.test(ac.x));
```

- We always need the parameters list, even when using anonymous classes to instantiate interfaces (which have no constructors themselves).
- When there's just one abstract method in the interface, you've got a *functional interface* and can do more with it (see below).

Advanced Interfaces

Interfaces can include or do all of the following:

- **extend other interfaces**; the child interface thus promises all of the methods from its parents and itself. Multiple interfaces may be extended by a single interface for a multiple inheritance situation:
  - interface I extends J, K, L { … }
- **static methods**. Since a static method doesn't need an object in order to be called, this is really no different than it being written in a class file, other than convenience's sake.
- **static final fields**. The modifiers `static` and `final` are assumed for all fields.
- **default methods**. Interfaces can provide inheritable implementations; both child interfaces and implementing classes obtain this version, but are all allowed to replace it.

When an interface inherits multiple versions of the same-signature method:

- if all versions were abstract, the interface may do either nothing or @Override to provide the one definition that gets used.
- if one or more were provided (defaults), it must `@Override` to arbitrate what gets called.
  - as part of its `default @Override` implementation, it can call a parent's version with:
    - `ParentName.super.methodName(args)`

Things in Classes

You can put just about anything in a class as a member, including the following:

- enumerations and interfaces can be placed inside a class.
- classes can be defined inside classes – called the enclosing class (outer) and the enclosed class (inner).
  - **static classes**: inside another class, with static modifier. Behaves as static member.
  - **inner classes**: inside another class, without the static modifier. Associated with one instance of the enclosing class, and may access its instance members.
  - **local classes**: defined inside a *method*. can access local (effectively-final) things.

Streams

We find ourselves, writing many repetitive patterns of code, such as loop-over-list, update-each-item, remove-items-that-fail-a-condition, combine-list-items-into-a-single-value. We can collapse these common blocks of code into a different function call if we get one new capability: the ability to pass a *chunk of code* in as a parameter. Since we can't pass in methods directly, Java needs use to pass in an object that contains the method we want to use. By adding some special syntax, and an extra bit of library code, Java lets us create anonymous class instances of single-function interfaces to accomplish this task.

- **Functional Interface**: any Java interface that has exactly one abstract method in it is a functional interface. It can be used like any other interface, but we may also create an anonymous
  @FunctionalInterface public interface Predicate<T>{ boolean test(T t); }
- **Lambda Expression**: special syntax that creates an anonymous class instance that implements a functional interface. Consists of: parameters list; arrow -> ; expression to be returned.
  Predicate<Integer> over5 = (Integer i) -> i>5 ;
- **Stream**: Any Collection or array can be converted to a stream. It represents a consumable sequence of values that might actually be infinite!
- Many operations on streams exist, needing many functional interface arguments.
  - some methods from **interface Stream<T>**:
    - **<R> Stream<R> map(Function<T,R> f)**. Given a Stream<T> and Function that can convert T values to R values, applies f to each item, generating a Stream<R>.
    - **Stream<T> filter(Predicate<T> p)**. Given a Stream<T> and a way to ask a question about each value, generate a Stream<T> of just the original stream's values that got a yes-answer.
    - **T reduce(T identity, BinaryOperator<T> accumulator)**. Starting with identity, combine each item from stream using the binary operator to get a single value. Example: Start with 0, keep using (+) to add numbers from a list to get the sum.
  - many other operations, functional interfaces, and Stream specializations exist!

Generic Wildcards, Bounded Wildcards

When we don't need to use a particular generic type, we can supply a ? symbol to fill the void (e.g., **List<?>**), and get away with less information needed. Just like regular type parameters, they can have upper and lower bounds. Here are some examples:

- **static <T> Pair<T> firsts(List<Pair<T,?>> pairs){**
  List<T> ans = new ArrayList<T>(); for (Pair<T,?> pair : pairs){ ans.add(pair.first()); } return ans;
  }
- wildcard as a supertype: Collections.addAll:
  - static <T> boolean addAll(Collection<? super T> c, T... elements)
- wildcard as a super- and sub-type in different places:
  - static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)
- wildcard as a super- and sub-type at once: Collections.sort, Collections.binarySearch:
  - static <T extends Comparable<? super T>> void sort(List<T> list)
  - static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)

Subtyping and Generics

- A common misconception arises when we combine generics and subtyping. Plugging in a parent type and child type into the same generic parameter does not create two subtyped things – for example, List<Child> is not a subtype of List<Parent>. Tracing this bit of code may help understand, as a Parent object would have been stored to a List<Child> if it were allowed.

```
List<Child> kids = new ArrayList<Child>();
List<Parent> rents = kids;  // doesn't compile! Our misconception might have us think otherwise.
rents.add(new Parent());    // rents aliases kids, so we have basically called kids.add(new Parent()).
```

Not Covered This Semester

Each semester we change focus a bit here and there. The following topics have been covered in previous semesters, but we will not be covering them this time. The course evolves a bit each semester.

Regular Expressions

- a way to describe a pattern within Strings that we are interested in finding.
- can be used to check if a given String matches, or contains, the pattern.
- we can also then extract different parts of the match for further calculations.
- Basics:
  - any character without special meaning matches itself. (This includes the space character!)
  - the period (.) matches any single character
  - repetition:
    - * : match zero or more of the previous thing.
    - + : match one or more of the previous thing.
    - ? : match zero or one of the previous thing.
    - {n} : match exactly n of the previous thing.
    - {n,m} : match between n and m inclusive of the previous thing.
    - {n,} : match n or more of the previous thing.
  - (parentheses): groups things together. (e.g., for use with the repetition meta-characters)
  - | : the vertical bar indicates "or", allowing the pattern on its left or right to be chosen. Can be used (with|many|choices|still|choosing|one|of|many).
  - character classes:
    - used to define how to match a **single** acceptable character. Just list them inside [ ]'s: [aeiou]
    - [a-z] : the dash(-) indicates a range on the ASCII chart; must be ascending.
    - [^abcde] : the caret (^), *only when it is the first symbol in a character class,* indicates "not the characters in this character class".
    - [nested[classes]] : union. (any single character from either can be selected).
    - [nested&&[classes]] : intersection. Only a character from both classes may be matched.
  - pre-defined groups: many common character classes have shortcut names.
    - \d : [0-9] (digits)
    - \s : [\t\n \f\r] (whitespace)
    - \w : [a-zA-Z0-9_] (identifier characters)
    - \D, \S, \W: [^0-9], [^\t\n \f\r], [^a-zA-Z0-9_] (non-whatever versions)
    - other special pre-defined groups also exist. (see Java Pattern class API if you're interested).
  - anchors: represent some property other than a specific character.

        - ^ : the beginning of line.
        - $: the end of line.
        - \b : a word-boundary, the point between a word-character \w and a non-word character \W.
        - \B : a non-word-boundary. (the point between 2 word-characters or 2 non-word characters).
    - embedded flags: ways to affect the overall meaning of a regular expression.
        - (?i) : case insensitive.
        - (?d) : unix newlines (only \n means newline)
        - (?m) : let ^, $ match on each line in the string, not just actual begin/end of entire string.
        - there are more of these… we just want to be comfortable with the basic idea.

- Representing Regular Expressions in Java.
    - We represent a regex inside of a Java String. This means that escaping characters becomes complicated.
        - any quote characters must be escaped (because it's in a String).
        - any occurrence of a backslash that was escaping something in the regular expression becomes a double backslash (so that the String contains the backslash character itself, and doesn't try to escape something).
        - regex's might already have a double backslash to represent a backslash character itself; it'll be a quadruple backslash in the String!
        - → good approach: write the regex normally first (perhaps in a comment in your source code), then represent each individual symbol in a Java String as necessary to obey Java String syntax.

- examples of methods we might use with regular expressions
    - `String` class
        - `boolean matches(String regex)`
        - `String replace (String target, String replacement)`
            - replaces ALL matches of the target string (which isn't a regex)
        - `String replaceAll (String regex, String replacement)`
            - replaces ALL matches of the target regex
        - `String[] split (string regexDelimiter)`
    - `Scanner` class
        - `String findInLine (String regex)`
        - `String findWithinHorizon (String regex, int horizon)`
        - `String next()` // relies upon the regex delimiter, settable with useDelimiter() method.

- Capture Groups
    - each parenthesized portion of a regular expression is a capture group. They are numbered 1 and up, found by scanning through the regex from left to right and numbering each opening parenthesis found. (The entire regex is considered group 0).
- Pattern and Matcher classes
    - In order to speed up loop bodies or use further functionality, we can use the Pattern and Matcher classes.
    - Pattern: allows us to represent a "compiled" regular expression as an object with useful methods, rather than just hiding it in a String.
        - `public static Pattern compile (String regex)`
        - `public Matcher matcher(String candidate)`
    - Matcher: Provides functionality related to checking for matches or finding the next match within a Pattern. Also provides the group method for extracting capture groups' values.
        - `public boolean matches()`

- checks for a complete match between the Pattern used to create this Matcher, and the candidate String supplied when the Matcher was created. If successful, calls to group with existing group #'s will succeed, returning the last match of that capture group.

- `public boolean find()`
  - checks for the first match within the candidate String. If successful, calls to group with existing group #'s will succeed, returning the last match of that capture group.
- `public String group(int g)`
  - if a successful match or find has been completed, and the regular expression that matched had a capture group of the given number input, then the String that corresponded to this group is returned.

Number Systems

- Different bases of interest: 10, 2, 16
- representations: just different symbols, with column values and symbol values combining to represent the full value.
  - representation choice (which base) doesn't change the value!
  - be comfortable counting up in bases 2, 10, 16 (chart from slides/labbook)
- conversions between bases:
  - towards base 10: 2 → 10, 16 → 10.
    - Process: find column values; find symbol values (their values in base 10, e.g. C is worth 12); multiply each symbol by its column value, add these results together.
  - from base 10: 10 → 2, 10 → 16
    - Process, version 1: find enough columns/their values until you've got enough space. Working from left to right, put as much as you can into each column. When you reach the right (the 1's column), you should have zero left over.
    - Process, version 2: keep dividing by the target base to get a quotient and remainder. The *remainder* of each division is the next column symbol from right-to-left, the quotient is what you divide by the base again. When quotient=0, you are done. (Watch out: remainder can be zero throughout, but you're not done until the quotient reaches 0.)