Java Intermediate

This doc gathers miscellaneous intermediate Java topics -- things we need for CS108 but which you may not have seen before in detail. The first part deals with many little topics, and the second part concentrates on Java Collections: List, Map, and Set. The 2nd part has different typography, as I'm preparing it in HTML to be published on the web.

Pervasive Shallowness in Java

- In C++, when you use an "=" for assignment, you always wonder how "deep" the copy is. In Java, = is always "shallow" -- copying the pointer but not the object. Suppose we have some Foo class:

```
Foo a = new Foo(1);
Foo b = new Foo(2);

a = b;   // shallow -- just make a point to 2nd Foo, GCing 1st Foo
bar(a);  // same, just pass a pointer into bar()
```

- It is typical in Java to have a few objects, and copy pointers to those few objects all over the place. We can afford to have pointers spread all over, since the GC figures out the deallocation for us.
- Can be a problem if we want to change an object, since many parts of the program may be pointing to it.
- This is not a problem if the object is "immutable" -- having no mutating methods, so it never changes once built.
- Another solution is to make a copy is if the class provides a "copy constructor" (C++ terminology) -- a constructor which takes an existing object as an argument, and creates a new object that is a copy. In general, making copies of things is something more often done in C++ than in Java. I think this is because the GC allows to not make copies in may situations.

```
Foo a = new Foo(1);
Foo copy = new Foo(a);  // make a copy of a
```

Array Review

- Ok, first remember how arrays work in Java:

```
int[] a = new  int[100];  // Allocate array in the heap, a points to the  array
a[0] = 13;
a.length       // access .length -- 100
a[100] = 13;   // out of bounds exception
```

"Arrays" Utility Class

- **Arrays** Class -- The Arrays class contains many static convenience methods that work on arrays -- filling, binary search, array equals, sorting.
- There is also a method in the System class, System.arraycopy(), that will copy a section of elements from one array to another (it also works correctly, copying elements within a single array). If the number of elements to copy is large, System.arraycopy() will probably be faster than your hand-written for-loop
- `System.arraycopy(source-array, source-index, dest-array, dest-index, length-to-copy);`

Arrays.equals(), deepEquals()

- The default "a.equals(b)" does not do a deep comparison for arrays, it just compares the two pointers. This violates the design principle of least surprise, but we're stuck with it for now.

- Use the static equals() in the Arrays class -- Arrays.equals(a, b) -- this checks that 1-d arrays contain the same elements, calling equals() on each pair of elements. For multi-dimensional arrays, use Arrays.deepEquals() which recurs to check each dimension.

Multidimensional Arrays

- An array with two or more dimensions is allocated like this...
  - int[][] grid = new int[100][100]; // allocate a 100x100 array
- Specify two indexes to refer to each element -- the operation of the 2-d array is simple when both indexes are specified.
  - grid[0][1] = 10; // refer to (0,1) element
- Unlike C and C++, a 2-d java array is not allocated as a single block of memory. Instead, it is implemented as a 1-d array of pointers to 1-d arrays. So a 10x20 grid has an "outer" 1-d array length 10, containing 10 pointers to 1-d arrays length 20. This detail is only evident if we omit the second index -- mostly we don't need to do that.

```
int temp;
int[][] grid = new int[10][20];  // 10x20 2-d array
grid[0][0] = 1;
grid[9][19] = 2;
temp = grid.length; // 10
temp = grid[0].length; // 20

grid[0][9] = 13;
int[] array = grid[0];  // really it's just a 1-d array
temp = array.length; // 20
temp = array[9]; // 13
```

- Note that System.arraycopy() does not copy all of a 2-d array -- it just copies the pointers in the outer 1-d array.

Packages / Import

Java Packages

- Java classes are organized into "packages"
- In this way, a class "Account" in your ecommerce package will not conflict with a class also named "Account" in the sales-tax computation package that you are using.
- Every java class has a "long" or "fully qualified" name the combines the class name and its package
- e.g. "String" full name is java.lang.String (that is, String is in the package "java.lang", the package that contains the most central classes of the language).
- e.g. "ArrayList" full name is java.util.ArrayList
- If you need to find the package of a class, you can look at its javadoc page -- the fully qualified name is at the top.
- In the compiled .class bytecode form of a class, the fully qualified name, e.g. java.lang.String, is used for everything. The idea of a "short" human readable name is a convention that is only used in the .java source files. All the later stages in the JVM use the full name.

Package Declaration

- A "package" declaration at the top of the .java source file indicates what package that class goes in
- package stanford.cslib; // statement at start of file
- If a package is not declared, the class goes into the one, catch-all "default" package. For simplicity, we will put our own classes in the default package, but you still need to know what a package is.

Compile Time Import

- "import java.util.*;" makes all the classes in that package available by their short names.

- "import java.util.ArrayList;" makes just that one class available
- When the compiler sees something like "Foo f = new Foo()", how does it know which "Foo" you are talking about?
- Can write it as "new java.util.Foo()" -- can always write the full name to disambiguate which class you mean
- Can add an "import java.util.*;" at the top of the file to make the short name work
- At compile time, the compiler checks that the referenced classes and methods exist and match up logically, but it does not link in their code. Each reference in the code, such as to "java.lang.String" is just left as a reference when the code is compiled. Later, when the code runs, each class (e.g. java.lang.String) is loaded when it is first used. Unlike in C, where you can statically link in a library, so its object code is copied into your program code.
- The ".*" version of import does not search sub-directories -- it only imports classes at that immediate level.
- Having lots of import statements will not make your code any bigger or slower -- it only allows you to use shorter names in your .java code.

"List" Example

- In the Java libraries, there are two classes with the name "List" -- java.util.List is a list data structure and java.awt.List is a graphical list that shows a series of elements on screen.
- Could "import java.util.*", in which case "List" is the util one. Or could import "java.awt.*", in which case "List" is the awt one. If both imports are used, then the word "List" is ambiguous, and we must spell it out in the full "java.util.List" form.
- In any case, the generated .class files always use the long java.util.List form in the bytecode.
- Compiling and referring to java.util.List does not link the java.util.List code into our bytecode. The java.util.List bytecode is brought in by the JVM at runtime. This is why your compiled Java code which many standard classes, is still tiny -- your code just has references to the classes, not copies of them.

Static

- Instance variables (ivars) or methods in a class may be declared "static".
- Regular ivars and methods are associated with objects of the class.
- Static variables and methods are not associated with an **object** of the class. Instead, they are associated with the **class itself**.

Static variable

- A static variable is like a global variable, except it exists inside of a class.
- There is a single copy of the static variable inside the class. In contrast, each regular instance variable exists many times -- one copy inside each object of the class.
- Static variables are rare compared to ordinary instance variables.
- The full name of a static variable includes the name of its class.
  - So a static variable named "count" in the Student class would be referred to as "Student.count". Within the class, the static variable can be referred to by its short name, such as "count", but I prefer to write it the long way, "Student.count", to emphasize to the reader that the variable is static.
- e.g. "System.out" is a static variable in the System class that represents standard output.
- Monster Example -- Suppose you are implementing the game Doom. You have a Monster class that represents the monsters that run around in the game. Each monster object needs access to a "roar" variable that holds the sound "roar.mp3" so the monster can play that sound at the right moment. With a regular instance variable, each monster would have their own "roar" variable. Instead, the Monster class contains a static Monster.roar variable, and all the monster objects share that one variable.

Static method

- A static method is like a regular C function that is defined inside a class.
- **A static method does not execute against a receiver object**. Instead, it is like a plain C function -- it can have parameters, but there is no receiver object.
- Just like static variables, the full name of a static method includes the name of its class, so a static foo() method in the Student class is called Student.foo().
- The Math class contains the common math functions, such as max(), sin(), cos(), etc.. These are defined as static methods in the Math class. Their full names are Math.max(), Math.sin(), and so on. Math.max() takes two ints and returns the larger, called like this: Math.max(i, j)
- A "static int getCount() {…" method in the Student class is invoked as Student.getCount();
- In contrast, a regular method in the Student class would be invoked with a message send (aka a method call) on a Student object receiver like s.getStress(); where s points to a Student object.
- The method "static void main(String[] args)" is special. To run a java program, you specify the name of a class. The Java virtual machine (JVM) then starts the program by running the static main() function in that class, and the String[] array represents the command-line arguments.
- It is better to call a static method like this: Student.foo(), NOT s.foo(); where s points to a Student object, although both syntaxes work.
  - s.foo() compiles fine, but it discards "s" as a receiver, using its compile time type to determine which class to use and translating the call to the Student.foo() form. The s.foo() syntax is misleading, since it makes it look like a regular method call.

static method/var example

- Suppose we have a Student class. We add a static variable and a static method for the purpose of counting how many Student objects have been created.
- Add a "static int numStudents = 0;" variable that counts the number of Student objects constructed -- increment it in the Student constructor. Both static and regular methods can see the static numStudents variable. There is one copy of the numStudents variable in the Student class, shared by all the Student objects.
- Initialize the numStudents variable with "= 0;" right where it is declared. This initialization will happen when the Student class is loaded, which happens before any Student objects are created.
- Add a static method getNumStudents() that returns the current value of numStudents.

```
public class Student {
    private int units;  // "units" ivar for each Student

    // Define a static int counter
    // to count the number of students.
    // Assign it an initial value right here.
    private static int numStudents = 0;

    public Student(int init_units) {
        units = init_units;

        // Increment the counter
        Student.numStudents++;
        // (could write equivalently as numStudents++)

    }

    public static int getNumStudents() {
        // Clients invoke this method as Student.getNumStudents();
        // Does not execute against a receiver, so
        // there is no "units" to refer to here
        return Student.numStudents;
    }

    // rest of the Student class
    ...
```

```
}
```

Typical static method error

- Suppose in the static getNumStudents() method, we tried to refer to the "units" instance variable...

```
public static int getNumStudents() {
    units = units + 1; // error
    return Student.numStudents;
}
```

- This gives an error message -- it cannot compile the "units" expression because there is no receiver object to provide instance variables. The error message is something like "cannot make static reference to the non-static 'units' ". The "static" and the "units" are contradictory -- something is wrong with the design of this method.
- Static vars, such as numStudents, are available in both static and regular methods. However, ivars like "units" only work in regular (non-static) methods that have a receiver object.

Files

File Reading

- Java uses input and output "stream" classes for file reading and writing -- the stream objects respond to read() and write(), and communicate back to the file system. InputStream and OuputStream are the fundamental superclasses.
- The streams objects can be layered together to get overall effect -- e.g. wrapping a FileInputStream inside a BufferedInputStream to read from a file with buffering. This scheme is flexible but a bit cumbersome.
- The classes with "reader" or "writer" in the name deal with **text files**
  - FileReader -- knows how to read text chars from a file
  - BufferedReader -- buffers the text and makes it available line-by-line
- For non-text data files (such as jpeg, png, mp3) use FileInputStream, FileOutputStream, BufferedInputStream, BufferedOutputStream -- these treat the file as a block of raw bytes.
- You can specify a unicode encoding to be used by the readers and writers -- defines the translation between the bytes of the file and the 2-byte unicode encoding of Java chars.

Common Text Reading Code

```
// Classic file reading code --  the standard while/readLine loop
// in a try/catch.
public void echo(String filename) {
    try {
        // Create reader for the given filename
        BufferedReader in = new BufferedReader(new FileReader(filename));

        // While/break to send readLine() until it returns null
        while (true) {
            String line = in.readLine();

            if (line == null) {
                break;
            }

            // do something with line
            System.out.println(line);
        }

        in.close();
    }
    catch (IOException except) {
        // The code above jumps to here on an IOException,
        // otherwise this code does not run.
```

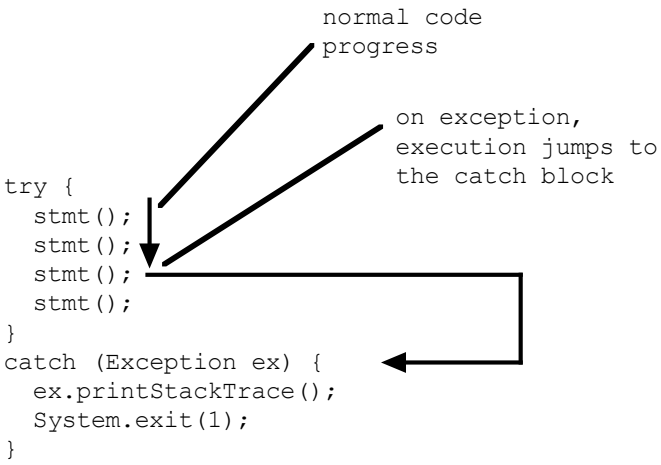
```
            // Good simple strategy: print stack trace, maybe exit
            except.printStackTrace();
            // System.exit(1);  // could do this too
        }

    }


// the while/readLine logic can be written as more compactly as
// "while ((line=in.readLine()) != null) {"
```

Exceptions

This is a basic introduction to exceptions. An exception occurs at runtime when a line of code tries to do something impossible such as accessing an array using an index number that is out of bounds of the array or dereferencing a pointer that is null.

An exception halts the normal progress of the code and searches for some error handling code that matches the exception. Most often, the error handling code will print some sort of warning message and then possibly exit the program, although it could take some more sophisticated corrective action.

Java uses a "try/catch" structure to position error-handling code to be used in the event of an exception. The main code to run goes in a "try" section, and it runs normally. If any line in the try section hits an exception at runtime, the program looks for a "catch-block" section for that type of exception. The normal flow of execution jumps from the point of the exception to the code in the catch-block. The lines immediately following the point of the exception are never executed.



For the file-reading code, some of the file operations such as creating the FileReader, or calling the readLine() method can fail at runtime with an IOException. For example, creating the FileReader could fail if there is no file named "file.txt" in the program directory. The readLine() could fail if, say, the file is on a CD ROM, our code is halfway through reading the file, and at that moment our pet parrot hits the eject button and flies off with the CD. The readLine() will soon throw an IOException since the file has disappeared midway through reading the file.

The above file-reading code uses a simple try/catch pattern for exception handling. All the file-reading code goes inside the "try" section. It is followed by a single catch-block for the possible IOException. The catch prints an error message using the built-in method printStackTrace(). The "stack trace" will list the exception at the top, followed by the method-file-line where it occurred, followed by the stack of earlier methods that called the method that failed.

It is possible for an exception to propagate out of the original method to be caught in a try/catch in one of its caller methods, however we will always position the try/catch in the same method where the exception first appears.

When your program crashes with an exception, if you are lucky you will see the exception stack trace output. The stack trace is a little cryptic, but it has very useful information in it for debugging. In the example stack trace below, the method hide() in the Foo class has failed with a NullPointerException. The offending line was line 83 in the file Foo.java. The hide() method was called by main() in FooClient on line 23.

```
java.lang.NullPointerException
    at Foo.hide(Foo.java:83)
    at FooClient.main(FooClient.java:23)
```

In production code, the catch will often exit the whole program, using a non-zero int exit code to indicate a program fault (e.g. call System.exit(1)). Alternately, the program could try to take corrective action in the catch-block to address the situation. Avoid leaving the catch empty -- that can make debugging very hard since when the error happens at runtime, an empty catch consumes the exception but does not give any indication that an exception happened. As a simple default strategy, put a printStackTrace() in the catch so you get an indication of what happened. If no exception occurs during the run, the catch-block is ignored.

In Java code, if there is a method call, such as in.readLine() above, that can throw an exception, then the compiler will insist that the code deal with the exception, typically with a try/catch block. This can be annoying, since the compiler forces you to put in a try/catch when you don't want to think about that case. However, this strict structure is one of the things that makes Java code so reliable in production. Aside: some exceptions such as NullPointerException or ArrayOutOfBounds are so common that almost any line of code can trigger them. These common exceptions are called "unchecked" exceptions, and code is not required to put in a try/catch for them.

Java 5 Generics

In my opinion, Java generics (added in version 5) are a mixed bag. Some uses of generics are simple to understand and make the code cleaner. I think they are clearly an improvement in the language. We will concentrate on these uses of generics. There are other, more complex uses of generics that I find hard to read, and I'm not convinced they are worth using at all.

Three Uses

- Here are the three use patterns of generics that I think work for most situations (examples below)…
- 1. Using a template class, like using ArrayList<String>
- 2. Writing template code with a simple <T> or <?> type parameter
- 3. Writing template code with a <T extends Foo> type parameter

Generic 1 -- Use Generic Class

- Many Java library classes have been made generic, so instead of just raw Object, they can be used in a way that indicates the type of object they hold.
- For example, the code below creates an ArrayList that holds Strings. Both the variable holding the pointer and the call to the constructor have <String> added.

```
ArrayList<String> strings = new ArrayList<String>();
strings.add("hi");
strings.add("there");
String s = strings.get(0); // no cast required!
```

- The advantage is that the compiler knows that the add() and get() methods take and return Strings, and so it can do the right type checking at compile time. We do not have to put in the (String) cast. The code that uses the list is now more readable. As a benefit of compile time typing, Eclipse code-assist now knows that strings.get(0) is a String and so can do code completion.
- The type of the iterator -- e.g. Iterator<String> -- must match the type of the collection.
- The plain type "List" without any generic type is known as the "raw" version. Raw versions still work in Java, and they essentially just contain pointers of type Object. You can assign back and forth between generic and raw versions, and it works, although it may give a warning.

  - e.g. Can store a List<String> pointer in a variable just of type List. This works, but gives a warning that it's better to keep it in List<String> the whole time.

- At runtime, all the casts are checked in any case, so if the wrong sort of object gets into a List<String> it will be noticed at runtime. This is why Java lets us mix between List<String> and raw List with just a warning-- it's all being checked at runtime anyway as a last resort.
- Eclipse tip: with the cursor where the <String> would go, hit ctrl-space. If Eclipse can deduce from context that a <String> or whatever is required, it puts it in for you, yay!

Boxing/Unboxing

- Normally, you cannot store an int or a boolean in an ArrayList, since it can only store pointers to objects. You cannot create an ArrayList<int>, but you can create an ArrayList<Integer>.
- With Java 5 "auto boxing", when the code needs an Integer but has an int, it automatically creates the Integer on the fly, without requiring anything in the source code. Going the other direction, auto unboxing, if the code has an Integer but needs an int, it automatically calls intValue() on the Integer to get the int value.
- This works for all the primitives -- int, char double, boolean, …
- This works especially well with generic collections, taking advantage of the fact that the collection type shows that it needs Integer or Boolean or whatever.

Unboxing Does Not Work With == !=

- Auto unboxing does not work correctly for == and !=.
- e.g. with two List<Integer> a and b, a.get(0) == b.get(0) -- does not unbox. Instead, it does an == pointer comparison between the two Integer objects, which is almost certainly not what you want.
- Use a.get(0).intValue() == b.get(0).intValue() to force the conversion to int. It works this way to remain compatible with the original definition of ==, but I hope this particular aspect of backward compatibility is dropped someday.

New Foreach Loop

- One the most likeable new features in Java 5 is a simple enhancement to the for-loop that iterates over any collection or array:

```
List<String> strings = ...

for (String s: strings) {
  System.out.println(s);
}
```

- This is a shorthand for looping over the elements with an iterator that calls hasNext()/next() in the usual way. The fancy iterator features -- such as remove() -- are not available.
- Nonetheless, this syntax is very handy for the very common case of iterating over any sort of collection.
- Design Lesson: if the clients of a system perform a particular operation very commonly (in this case, iterating over all the elements in a collection) -- it's a good design idea to have a simple facility that makes that common case very easy. It's ok if the facility does not handle more advanced uses. It can be simple and focus just on the common case. Putting in such a special-purpose facility is, in a way, inelegant -- it creates more than one way to do things. However, experience shows that making common cases very easy is a good design idea.

Generic List Example Code

```
// Shows basic use of the ArrayList and iterators
// with the generics.
public static void demoList() {
    // ** Create a List<String>
    List<String> a = new ArrayList<String>();
    a.add("Don't");
    a.add("blame");
    a.add("me");
```

```
    // ** foreach -- iterate over collection easily
    for (String str: a) {
        System.out.println(str);
    }

    // ** Instead of Iterator, make an Iterator<String>
    Iterator<String> it = a.iterator();
    while (it.hasNext()) {
        // NOTE: no cast required here -- it.next() is a String
        String string = it.next();
        System.out.println(string);
    }

    // ** Likewise, can make a List<Integer>
    List<Integer> ints = new ArrayList<Integer>();
    for (int i = 0; i < 10; i++) {
        ints.add(new Integer(i * i));
    }

    // No casts needed here -- it knows they are Integer
    int sum = ints.get(0).intValue() + ints.get(1).intValue();

    // With auto Unboxing, can just write it like this...
    sum = ints.get(0) + ints.get(1);

    // Can go back and forth between typed Collections and untyped "raw"
    // forms -- may get a warning.
    List<String> genList = new ArrayList(); // warning
    List rawList = new ArrayList<String>(); // no warning
    rawList.add("hello"); // warning
    genList = rawList; // warning
    rawList = genList; // no warning
```

Generic Map Example Code

```
public static void demoMap() {
    // ** Make a map, specifying both key and value types
    HashMap<Integer, String> map = new HashMap<Integer, String>();

    // Map Integers to their words
    map.put(new Integer(1), "one");
    map.put(new Integer(2), "two");
    map.put(3, "three"); // Let the autoboxing make the Integer
    map.put(4, "four");

    String s = map.get(new Integer(3)); // returns type String
    s = map.get(3); // Same as above, with autoboxing
    // map.put("hi", "there");  // NO does not compile

    // ** Auto unboxing -- converts between Integer and int
    Integer intObj = new Integer(7);
    int sum = intObj + 3; // intObj unboxes automatically to an int, sum is 10

    // ** More complex example -- map strings to lists of Integer
    HashMap<String, List<Integer>> counts = new HashMap<String, List<Integer>>();

    List<Integer> evens = new ArrayList<Integer>();
    evens.add(2);
    evens.add(4);
    evens.add(6);
    counts.put("evens", evens);

    // Get the List<Integer> back out...
    List<Integer> evens2 = counts.get("evens");
    sum = evens2.get(0) + evens2.get(1); // unboxing here, sum is 6
}
```

Generic 2 -- Define a Generic <T> Class

- You can define your own class as a generic class. The class definition code is parameterized by a type, typically called <T>. This is more or less what ArrayList does. At the very start of the class, the parameter is added like this:
  `public class Foo<T> { ...`
- In the simplest case, the class code doe not have any specific constraint on what type T is. Inside the class, T may be used in limited ways:
  - declare variables, parameters, and return types of the type T
  - use = on T pointers
  - call methods that work on all Objects, like .equals()

Generic <T> Operations, Erasure

- In the source code implementing the generic class, T may be used to declare variables and return types, and may be used to declare local generic variables, such as a List<T>. Essentially, T behaves like a real type such as String or Integer.
- At a later stage in the compilation, T is replaced by Object. So at run-time, the notion of T is gone -- it's just Object by then. This is known as the "erasure" system, where T plays its role to check the sources early in compilation, but is then erased down to just plain Object.
- Remember: where you see "T", it is just replaced by "Object" to produce the code for runtime.
- Therefore things that need T at runtime do not work -- e.g. creating a "T" array with code like: "new T[100]". This cannot work -- at runtime, there is no T, just Object.
- The erasure system provides basic generic type checking at compile time. The erasure approach is a compromise that adds generics to Java while remaining compatible with pre Java 5 systems. It is a workable compromise.

Generic <T> Pair Example (add List<T>)

```
/*
 Generic <T> "Pair" class contains two objects, A and B, and supports
 a few operations on the pair. The type T of A and B is a generic
 parameter.

 This class demonstrates parameterization by a <T> type.
 The class does not depend on any feature of the T type --
 just uses "=" to store and return its pointers. The T type can be used
 for variables, parameters and return types. This approach is quite easy to do,
 and solves all the cases where we don't really care about the T type.
 This is basicaly how ArrayList and HashMap do it.
*/

public class Pair<T> {
    private T a;              // Can declare T variables
    private T b;
    private List<T> unused;  // Can use T like this too

    public Pair(T a, T b) {
        this.a = a;
        this.b = b;
    }

    public T getA() {
        return a;
    }

    public T getB() {
        return b;
    }

    public void swap() {
        T temp = a;  // NOTE T temporary variable
        a = b;
        b = temp;
```

```
    }

    // True if a and b are the same
    public boolean isSame() {
        return a.equals(b);
        // NOTE Can only do things on T vars that work on any Object
    }

    // True if a or b is the given object
    // NOTE: use of T as a parameter
    public boolean contains(T elem) {
        return (a.equals(elem) || b.equals(elem));
    }

    public static void main(String[] args) {
        // ** Make a Pair of Integer
        Pair<Integer> ipair = new Pair<Integer>(1, 2);
        Integer a = ipair.getA();
        int b = ipair.getB();  // unboxing

        // ** Also make a Pair of String
        Pair<String> spair = new Pair<String>("hi", "there");
        String s = spair.getA();
    }

    // Show things that do not work with T
    private void doesNotWork(Object x) {
        // T var = new T();  // NO, T not real at runtime -- erasure

        // T[] array = new T[10];  // NO, same reason

        // T temp = (T) x;  // NO, same reason (like (Object) cast)
    }
}
```

Generic <T> Method

- Rather than making a whole class generic, the generic syntax can work on a single method. The syntax is that the <T> goes before the return type: `public <T> void foo(List<T> list) {`
- Here is an example where T allows the method to work for any sort of List:

```
// <T> Method -- use a <T> type on the method to
// identify what type of element is in the collection.
// The <T> goes just before the return type.
// T can be used to declare variables, return types, etc.
// This is ok, but slightly heavyweight, since in this case
// we actually don't care what type of thing is in there.
// This removes elements that are == to an adjacent element.
public static <T> void removeAdjacent(Collection<T> coll) {
    Iterator<T> it = coll.iterator();
    T last = null;
    while (it.hasNext()) {
        T curr = it.next();
        if (curr == last)
            it.remove();
        last = curr;
    }
}
```

Generic <?> Method

- The <?> generic parameter is like a <T> parameter but a little simpler. The "?" is a wildcard which I think of as meaning "don't care" -- a type that can be anything, since the code does not care what it is.
- Here is a bar() that uses "?" to take a list of anything:

```
void bar(List<?> list) {
```

- We can use the list inside bar(), but we cannot assume anything about what type it contains -- we don't even have a name for the type it contains. Note that there is no <?> at the start of the method. The <?> only appears in the parameter list.
- We can make local generic variables that mention the <?>, such as:

```
List<?> temp = list;
Iterator<?> it = list.iterator();
```

- We cannot declare a variable of type "?", however "Object" works fine:

```
Object elem = list.get(0);
```

- I like the <?> variant to encode true "I don't care" cases. It has the simplest syntax and the fewest features.

```
// <?> Method -- Use a "wildcard" <?> on the parameter, which
// is basically a "don't care" marker. The <?> is not listed
// before the return type.
// We cannot declare variables of type ?, but we can use Object
// instead.
// We can declare local generic variables that mention <?>.
public static void removeAdjacent2(Collection<?> coll) {
    Iterator<?> it = coll.iterator(); // Can use <?> in local variables
    Object last = null; // Cannot use ? as a var type, use Object
    while (it.hasNext()) {
        Object curr = it.next();
        if (curr == last)
            it.remove();
        last = curr;
    }
}
```

Passing Any Sort of List -- List<Object> vs. List<T>

- Suppose we want a method that takes as a parameter any sort of list -- list of Strings, list of Integers etc.
- At first glance, it looks like a parameter of type List<Object> could work as a catch-all for any sort of list. However, that does not work, because of the container anomaly described below.
- That is why List<T> and List<?> exist -- they are the way to encode that you take a List (or whatever) and it can contain any type.

Generic 3 -- ?/T with "extends" Generics

- The plain <?>/<T> generics doesn't let us assume anything about T. The "extends" form allows us to add a constraint about T.
- For example, with a <T extends Number> -- for any T value, we can assume it is a Number subclass, so .intValue() etc. just work. This is known as a "bounded wildcard".
- This technique can be used on a whole class, such as a Pair that contains numbers:

```
public class PairNumber <T extends Number> {
    private T a;
    private T b;

    public PairNumber(T a, T b) {
        this.a = a;
        this.b = b;
    }

    public int sum() {
        // Note: here we can use Number messages, like intValue().
        return (a.intValue() + b.intValue());
    }
    ...
```

?/T Extends Method

- The "extends" constraint also works on ?/T methods

```
// ?-Extends -- rather than Collection<?>, the "extends Number"
// adds a constraint that the elements must be subclasses of Number.
// The Number subclasses include Integer, Double, Long, etc.
// Returns the int sum of all the numbers in the collection.
public static int sumAll(Collection<? extends Number> nums) {
    int sum = 0;
    for (Number num : nums) {
        sum += num.intValue(); // All the Number types respond to intValue().
    }
    return sum;
}
```

Container Anomaly

- Why must we use List<T> to for a list of any type, rather than just List<Object>?
- There is a basic problem between the type of a container and the type of thing it contains.
- A pointer to a String (the subclass) can be stored in a variable of type Object (the superclass). We do that all the time.
- However, a pointer to a List<String> cannot be stored in a variable type List<Object> -- this is unintuitive, but it is a real constraint. The reason for this constraint is demonstrated below.

Container Anomaly Example Code

```
    // say we have this method
    public static void doSomething(List<Object> lo) {

    }

    public static void problem() {
        Object o;
        String s = "hello";
        o = s;  // fine

        List<Object> lo = new ArrayList<Object>();
        List<String> ls = new ArrayList<String>();

        // lo = ls;  // NO does not compile

        // The following would be a big prob if above were allowed.
        // If we allow ls to be in lo, we lose track of the constraint
        // that ls should only contain Strings.
        lo.add(new Integer(3));

        // doSomething(ls);  // NO same problem

        // The point: you can assign a pointer of type sub to a pointer of type
        // super. However, you cannot assign a pointer type container(sub) to a
        // pointer of type container(super).
        // Therefore Collection<Object> will not work as a sort of catch-all
        // type for any sort of collection.
    }
```