**The Big List**

1. [Generics](http://www.cs.indiana.edu/classes/jett/sstamm/#_generics)
2. [For/in loop](http://www.cs.indiana.edu/classes/jett/sstamm/#_forin)
3. [Autoboxing/Unboxing](http://www.cs.indiana.edu/classes/jett/sstamm/#_boxing)
4. [Typesafe Enums](http://www.cs.indiana.edu/classes/jett/sstamm/#_enums)
5. [Varargs](http://www.cs.indiana.edu/classes/jett/sstamm/#_varargs)
6. [Static Import](http://www.cs.indiana.edu/classes/jett/sstamm/#_staticimport)
7. [Annotations (Metadata)](http://www.cs.indiana.edu/classes/jett/sstamm/#_annotations)

**1.1 Generics**

Generics provide a way to create and use type safe data structures. This means that no longer do you have to create a List of the basic Objects then typecast every time you pull stuff out! You can declare your list to automatically typecast the stuff inside:

List things = createListOfBorkObjects();

for(Iterator i = things.iterator() ; i.hasNext() ; ) {

Bork item = (Bork)i.next();

//do something useful

}

Simply becomes...

List<Bork> things = createListOfBorkObjects();

for(Iterator<String> i = things.iterator() ; i.hasNext() ; ) {

Bork item = i.next();

//do something useful

}

The Java compiler also protects things from having non-Bork objects inside. If you try to put a String or anything else in, you'll get an error. This essentially means that you can create a list with specific types now instead of just objects. These exist in other classes such as Map which uses two:

Map<String, Bork> myMap = new HashMap<String, Bork>();

That creates a map that uses a String key and a Bork value. This implementation kind of looks like something using templates in C++...

*Be careful though!* You can create a string iterator that iterates over a list of non-strings that will only become an error at runtime. The compiler doesn't catch this.

Also, sometimes when using a type that can be parameterized, but not specifying the parameter type (in angle brackets) you can get **lint** warnings. This just means that you haven't provided a specific type, and the current definition of the type is "fuzzy" -- like lint, get it?

#### 1.2 For/in loop

This is probably one of the coolest new features. I personally hate using iterators--it seems redundant and annoying sometimes. Well, lucky for me there's a way around them now!

So now you can do this:

for(Iterator lineup = list.iterator() ; lineup.hasNext() ; ) {

Object thatThing = lineup.next();

myMonster.eat(thatThing);

}

In a shortened:

for(Object thatThing : list) {

myMonster.eat(thatThing);

}

Much prettier, no? This works with arrays too.

int[] nums = { 1, 2, 3, 4, 5, 6 };

for(int n : nums) {

System.out.println(n);

}

Say you want to get your class to work with this nifty loop. Then, you have to implement Iterable (or extend something that does). This involves telling Iterable what type of things you iterate over. You can define a custom iterator to do something more robust, but for this illustration, I'm just going to grab one out of the list.

public class MailBox implements Iterable<MailMessage> {

/\*\* structure storing the messages \*/

private ArrayList<MailMessage> messages;

//...

/\*\*

\* Implemented for Iterable.

\*/

public Iterator<MailMessage>() {

return messages.iterator();

}

//...

}

For more detailed information, see Java 1.5 Tiger: A Developer's Notebook[[4](http://www.cs.indiana.edu/classes/jett/sstamm/" \l "ref:book)] or the information on Sun's J2SE 5.0 language documentation.

#### 1.3 Autoboxing/Unboxing

Integer i = new Integer(4);

int j = i.intValue();

Number n = new Float(3.14159);

Boolean stuff = new Boolean(false);

// stuff before ? must be a boolean (lower case)

System.out.println( stuff.booleanValue() ? "Yep" : "Nope" );

Sick of this? Me too. Do this instead:

Integer i = 4;

int j = i;

Number n = 3.14159f;

Boolean stuff = false;

System.out.println( stuff ? "Yep" : "Nope" );

This is pretty nice. Especially since you can use ++ and other similar operators with the wrapper types now too.

#### 1.4 Typesafe Enums

Enums are just magic classes to help prevent the method less-interface ant pattern. They let you make classes that will enumerate values, but also keep the types specific. Before, we could simulate enums with a bunch of static final int variables or something. The problem with those is that you could confuse **any** int with one of the constants. With enumerations, only the values in the enum are valid. For example:

public enum JettStaff {

ADRIAN,

ARIJIT,

BETH,

ERIC,

KATIE,

KATY,

RAJA,

RICH,

SUZANNE

};

JettStaff x = JettStaff.SUZANNE;

Now, it gets even cooler. I don't have to keep track of separate information to store, say the peoples' full names. I can associate them directly, just like in a class! Each of the values of JettStaff are instances of the JettStaff enumeration, so we can define a constructor and a toString() method.

public enum JettStaff {

ADRIAN("Adrian German"),

ARIJIT("Arijit Sengupta"),

BETH("Beth Plale"),

ERIC("Eric Wernert"),

KATIE("Katie A. Siek"),

KATY("Katy Borner"),

RAJA("Raja Sooriamurthi"),

RICH("Rich Kick"),

SUZANNE("Suzanne Menzel");

private String name;

public JettStaff(String n) { this.name = n; }

public String toString() { return this.name; }

}

JettStaff x = JettStaff.SUZANNE;

System.out.println(x);

*But wait, it gets cooler!* Now you can also give each enumerated value a custom body. Since they're each instances, you could design a toString() method for each:

public enum JettStaff {

ADRIAN("Adrian German") {

public String toString() {

return name + " (dgerman@indiana.edu)";

}

},

ARJIT("Arjit Sengupta") {

public String toString() {

return name + " (asengupt@indiana.edu)";

}

},

// and on for the rest...

private String name;

public JettStaff(String n) { this.name = n; }

}

JettStaff x = JettStaff.SUZANNE;

System.out.println(x);

Last but not least, enums can extend each other. Imagine that!

#### 1.5 Varargs

What is your impression of "..."? It's a nice little note that might come in handy. Notice how when you pass arguments from the command line ...

C:/> java MyProg a b c

You gave me 3 args! Yay.

...

... you don't have to pass an array of stuff. The runtime automatically converts the arguments into an array of strings. You can do that now in all of your methods! For example, instead of doing this:

public class VarArgs {

public static void main(String[] args) {

String[] newArgs = {"a", "b", "c"};

vaMethod(newArgs);

}

public void vaMethod(String[] args) {

System.out.println("You gave me " + args.length + " args! Yay.");

}

}

You can declare it more easily, and not have to construct the array ahead of time:

public class VarArgs {

public static void main(String[] args) {

vaMethod("a", "b", "c");

}

public void vaMethod(String... args) {

System.out.println("You gave me " + args.length + " args! Yay.");

}

}

Notice that when you use the ... syntax, it automatically treats that parameter as an array but you don't have to pass it in that way. Nifty. Let's add one of those for/in loops:

public class VarArgs {

public static void main(String[] args) {

vaMethod("a", "b", "c");

}

public void vaMethod(String... args) {

for(String s : args)

System.out.println(s);

}

}

#### 1.6 Static Import

Remember making all those interfaces that just have constants in them?

import java.awt.\*;

public interface BouncingBallConstants {

public static final Color BACK\_COLOR = Color.WHITE;

public static final Color BALL\_COLOR = Color.BLUE;

public static final int BALL\_SIZE = 50;

}

Scrap that. Put these into a **REAL** class and then just import the static members from all the other ones using import static <Package or Class>;. This addition is pretty straightforward, and it helps prevent bloat and the "methodless interface" design antipattern.

#### 1.7 Annotations (Metadata)

Now for the weirdest part. Annotations are not really something that will affect how you program in Java, unless you need to associate some sort of *metadata* or annotations with classes, methods, variables, etc.

So what are annotations anyway? That's a good question. They provide a little extra information about the classes you write, and a class can use the Reflection package later to read the annotations. These are useful because you can attach extra information to your code that may determine how it is used or maybe if it is used at all.

For example, in J2SE 5, you can *declare* your intent to override a method like toString() in one of your classes:

public class MyClass extends Object {

@Override

public String toString() {

return "My overridden method!";

}

}

In the above example, we declare that we will override the immediately following toString() method. So the compiler looks in our superclass (Object) for the same metho and makes sure it exists. If for some reason we had *overloaded* toString() by declaring it with different parameters and maybe return type, then the compiler would throw an error saying we didn't override correctly. This is really useful if you want to make sure you override a method as opposed to simply over*loading* it.

Of course you can define your own annotations. They're basically like interfaces, but they can contain values. An example annotation looks like:

public @interface Conference {

String what();

String when();

String location();

}

This annotation declares three members: what, when, location and sets them up to have "getters" and "setters" automatically! That means each @Conference annotation has those three fields associated with it, and I don't have to define the accessor and mutator methods to set them up (see the next code listing). If I define this annotation like this, I can use it to mark code that I use for the Jett conference:

@Conference(what="JETT",

when="November 2004",

location="IUB")

public class MyMagicJettClass {

//...

}

And now the @Conference type of data is associated with my class. Later on, I could write an analyzer that goes through all of my code and lets me know which classes were used at conferences as well as which conferences they were used at and when. This specific example doesn't have any effect on the way MyMagicJettClass operates.

So the annotations require two-fold cooperation: the programmer must properly annotate her code to provide adequate metadata needed and other developers who will want to know this metadata must know how to extract it using the Java Reflection package. That's a whole hours-long session on how to extract them, so I'm not going to go into depth here.

Where are they useful? Say you are working with RMI (Remote Method Invocation) and you don't want all of your methods available remotely. You could annotate the remotable ones with a @Remoteannotation, then whatever serves up the remote access can only allow those to be remotely accessed. There are a ton of great uses for these, and they are fully extendable (you can annotate annotations)!

### 2. How it will affect you NOW

All of the previous Java language features will still work. Some are deprecated, but not very many. You should also know that the 2004 exam was written before 1.5 was available, so its features are not at all critical.

The topics covered by the exam don't include generic typing, annotations/metadata or variable arguments (since really they're just language features and not so much programming constructs). This is good news, since if your students don't immediately master these new features it won't harm them.

What if my students use new 1.5 language features on the exam? If they're used properly (and the code is well-designed), the readers will most likely not deduct any points. But in the end the deductions are up to the readers. Automatic boxing and unboxing as well as correct use of the for/in loop will be acceptable.

The biggest problem would be a student who tries to use one of these new features (such as generics or enums) but misuses them terribly, obscuring the code. Then a reader might not at all understand what constructs the student was trying to use and mistake it for non-code garbage. Because the readers are as new to the 1.5 features as we are, we cannot expect them to have mastered everything to the point of being able to identify misuse of one.