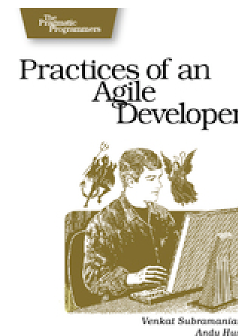
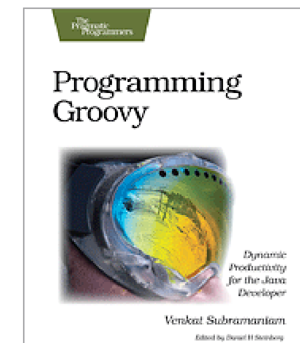
KNOW YOUR JAVA?

```
speaker.identity {
  name        'Venkat Subramaniam'
  company     'Agile Developer, Inc.'
  credentials 'Programmer', 'Author', 'Trainer'
  blog        'http://agiledeveloper.com/blog'
  email       'venkats@agiledeveloper.com'
}
```





Abstract

- Java has been around for well over a decade now. It started out with the goal of being simple. Over the years, its picked up quite a bit of features and along comes complexity. In this presentation we will take a look at some tricky features of Java, those that can trip you over, and also look at some ways to improve your Java code. Java features. Set of tricks. Tips to improve your Java code.

Creating Strings

```
// Which of these two is better?
String str1 = new String("Hello");
String str2 = "Hello";
```

Creating Strings

- Strings are immutable
- `"Hello"` is already an instance of String
- using `new` on String creates a new redundant instance
- Using new to create String is not a good idea

Comparing Strings

```
public static void compare(String s1, String s2)
{
  // Which of these two is better?
  System.out.println(s1 == s2);
  System.out.println(s1.equals(s2));
}
```

Comparing Strings

- == compares identity—don't use it to compare Strings
- `.equals()` compares value

```
compare("hello", "hello");
compare("hello", new StringBuffer("hello").toString());
```

```
true
true
false
true
```

Concatenating Strings

```
// Which of these two is better
System.out.println(s1 + s2);

System.out.println(new StringBuffer(s1).append(s2));
```

Concatenating Strings

* They both do almost the same thing under the hood
* If all you're doing is concatenating two Strings occasionally like above, first one is sufficient and even better
* If you're concatenating large number of Strings, then consider alternatives

Concatenating Strings

```
// Which of these is better?
String s = "";
for(int i = 0; i < INDEX; i++)
{
    s += "."; // or whatever
}

StringBuffer b = new StringBuffer(INDEX);
for(int i = 0; i < INDEX; i++)
{
    b.append("."); // or whatever
}
```

Concatenating Strings

* Neither one
* If thread synchronization is not needed, use StringBuilder instead
* Go ahead and measure time taken for concatenation for each option
* Remember to allocate enough size for StringBuilder

```
StringBuilder sb = new StringBuilder(INDEX);
for(int i = 0; i < INDEX; i++)
{
    sb.append("."); // or whatever
}
```

Object finalize

```
public class Test
{
    private FileWriter writer;

    public Test() throws Exception { writer = new FileWriter("output.txt"); }

    protected void finalize() throws Throwable { super.finalize(); writer.close(); }

    public void info(String msg) throws Exception { writer.write(msg); }

    public static void main(String[] args) throws Exception
    {
        // What's wrong with this code?
        Test obj = new Test();
        obj.info("test");
    }
}
```

Object finalize

- File is not closed, content not written (flushed)
- Resource may be held much longer than needed
- don't rely on finalizers
- Option 1: provide your own clean up method

```
public void close() throws IOException { writer.close(); }

public static void main(String[] args) throws Exception
{
    Test obj = new Test();
    obj.info("test");
    obj.close();
}
```

There's still a problem here...

Object finalize

- In Option 1, you're under the mercy of caller to call close()
- Option 2: Try making it automatic
- When Java gets closure, this will become easier, but until then...

```
interface Runner { public void run(Test t) throws Exception; }
private Test() throws Exception { writer = new FileWriter("output.txt"); }
private void close() throws IOException { writer.close(); }
public static void use(Runner r) throws Exception
{
    Test obj = new Test();
    r.run(obj);
    obj.close();
}

public static void main(String[] args) throws Exception
{
    Test.use(new Runner()
    {
        public void run(Test t) throws Exception
        {
            t.info("test");
        }
    });
}
```

There's still a problem here...

Resource closed automatically...

Execute Around Method Pattern

Cleanup

```
//What's wrong with this code?
Test obj = new Test();
r.run(obj);
obj.close();
```

Cleanup

* If exception is thrown, close() may not be called...
* Wrap in try-finally

```
Test obj = new Test();
try
{
  r.run(obj);
}
finally
{
  obj.close();
}
```

equals

```
class Dog
{
  private int _age;
  public Dog(int age) { _age = age; }
  public boolean equals(Object other)
  {
    if (other == this) return true;
    if (other instanceof Dog)
    {
      Dog otherDog = (Dog) other;
      if (_age == otherDog._age) return true;
    }
    return false;
  }
}
```

```
class Dalmation extends Dog
{
  private int _spots;
  public Dalmation(int age, int numberOf
  {
    super(age);
    _spots = numberOfSpots;
  }
  public boolean equals(Object other)
  {
    // What's wrong?
    if (!super.equals(other)) return false;
    if (other instanceof Dalmation)
    {
      Dalmation otherDalmation = (Dalmation) other;
      if (_spots == otherDalmation._spots) return true;
    }
    return false;
  }
}
```

equals

* equals() must be symmetric, reflective, and transitive

```
public class Sample
{
  public static void main(String[] args)
  {
    Dog dog1 = new Dog(2);
    Dog dog2 = new Dalmation(2, 250);

    System.out.println(dog1.equals(dog2)); // => true
    System.out.println(dog2.equals(dog1)); // => false
    // Lacks symmetry
  }
}
```

equals

* ensure you're checking objects of the right type

```
public boolean equals(Object other)
{
  if (other == this) return true;
  if (other.getClass() == getClass())
  {
    Dog otherDog = (Dog) other;
    if (_age == otherDog._age) return true;
  }
  return false;
}
```

May be a problem if objects in different containers/classloaders. May want to check for their actual class name?

equals...

```
// What's missing?
class Car
{
  private int _year;
  private int _miles;
  public Car(int year, int miles) { 

  public boolean equals(Object other)
  {
    if (this == other) return true;
    if (getClass() == other.getClass())
    {
      Car otherCar = (Car) other;
      if (otherCar._year == _year && otherCar._miles == _miles)
      {
        return true;
      }
    }

    return false;
  }
}
```

equals...

- Equal objects are required to have equal hashCodes

```
java.util.HashMap<Car, String> cars = new java.util.HashMap<Car, String>();
cars.put(new Car(2008, 25), "JKV 28V");

System.out.println(cars.get(new Car(2008, 25))); //=> null
```

equals...

* If you override equals, provide hashCode() method

```
public int hashCode()
{
  return 7 * _year + 11 * _miles;
  // choose some prime numbers as multiplicator
  // Make sure for equal objects hashCode is equal
  // Would be nice for hashCode to be different for objects that are not equal
}
```

static in Generics

```
class MyList<T>
{
    public static int count;

    public MyList() { count++; }

    public int getCount() { return count; }
}

public class Test
{
    public static void main(String[] args)
    {
        // What's the output?
        MyList<Integer> list1 = new MyList<Integer>();
        MyList<Integer> list2 = new MyList<Integer>();
        MyList<Double> list3 = new MyList<Double>();

        System.out.println(list1.getCount());
        System.out.println(list2.getCount());
        System.out.println(list3.getCount());
    }
}
```

static in Generics

- Type erasure converts T to Object
- All instances of Generics will share the same static
- Don't use static in Generic classes
- This becomes clearer if you convert getCount to static and call it as MyList<Integer>.getCount()
- Compiler allows only MyList.getCount()

static and Inheritance

```
class Base
{
    public static String info() { return "hello Base"; }
}

class Derived extends Base
{
    public static String info() { return "hello Derived"; }
}

public class Test
{
    public static void main(String[] args)
    {
        Derived d = new Derived();
        System.out.println(d.info());
        Base b = d;
        System.out.println(b.info());
    }
}
```

static and Inheritance

* Output from above code is

```
hello Derived
hello Base
```

* static methods are not polymorphic
* Don't call static methods on instances
* Avoid confusion, only call them on classes

```
System.out.println(Base.info());
System.out.println(Derived.info());
```

static and Threading

```
class Creature
{
    public static int count;

    public Creature()
    {
        // What's wrong with this code?
        count++;
    }

    public static int getCount() { return count; }
}
```

static and Threading

- count++ is not atomic
- If multiple threads invoke constructor at same time, count will be in jeopardy
- Does this fix?

```
public Creature()
{
    //fixed?
    synchronized (this)
    {
      count++;
    }
}
```

static and Threading

- Nope, not effective synchronization
- How about?

```
public Creature()
{
    //How about?
    synchronized (Creature.class)
    {
      count++;
    }
}
```

static and Threading

* Too sweeping
* Generally, synchronizing on Class is not a smart idea
* If another part of code synchronizes on the same, it limits concurrency
* Make synchronization very specific and narrow

```
public static int count;
private final static Object countLock = new Object();

public Creature()
{
    //Much better in general, but for this...
    synchronized (countLock)
    {
      count++;
    }
}
```

static and Threading

- For simple increment operation, use AtomicLong

```
public static AtomicLong count = new AtomicLong();

public Creature()
{
    count.incrementAndGet();
}
```

Overriding

```java
class Base
{
    public void foo(double a)
    {
        System.out.println("Base.foo(double)");
    }
}

class Derived extends Base
{
    public void foo(int a)
    {
        System.out.println("Derived.foo(int)");
    }
}
```

```java
//What's the output?
Derived d = new Derived();
Base b = d;
b.foo(5);
d.foo(5);
```

Overriding

* foo in Derived is hiding foo in Base
* Don't override and change signature

```
Base.foo(double)
Derived.foo(int)
```

Overriding

```
public class Test                                  //Test.groovy
{
    public static void main(String[] args)
    {
        //What's the output?
        Derived d = new Derived();
        Base b = d;
        b.foo(5);
        d.foo(5);
    }
}

class Base
{
    public void foo(double a)
    {
        System.out.println("Base.foo(double)");
    }
}

class Derived extends Base
{
    public void foo(int a)
    {
        System.out.println("Derived.foo(int)");
    }
}
```

```
Derived.foo(int)
Derived.foo(int)
```

- Not all languages (on the JVM) are the same
- Try running the above Java code through Groovy!
- Groovy's multimethods produces a different result

Polymorphism

```
class Employee
{
    public Employee() { work(); }

    public void work() { System.out.println("Do work..."); }
}

class Window { public void open() { } }

class Manager extends Employee
{
    private Window window = new Window();

    public void work()
    {
        window.open();
        super.work();
    }
}
```

```
public static void main(String[] args)
{
    //What's the output?
    System.out.println(new Manager());
}
```

Polymorphism

- Don't call polymorphic (non-final) methods from within constructors

```
Exception in thread "main" java.lang.NullPointerException
```

Generics and Interfaces

```
private static void playWith1()
{
    ArrayList<Integer> list = new ArrayList<Integer>();
    list.add(1);
    list.add(2);
    list.add(3);
    System.out.println(list.size());
    list.remove(0);
    System.out.println(list.size());
}

private static void playWith2()
{
    Collection<Integer> list = new ArrayList<Integer>();
    list.add(1);
    list.add(2);
    list.add(3);
    System.out.println(list.size());
    list.remove(0);
    System.out.println(list.size());
}
```

```
//What's the output?
playWith1();
playWith2();
```

Generics and Interfaces

- ArrayList's remove looks for index
- Collection's remove looks for object
- So, it converts 0 to Integer (which does not exist in list)
- Try running the above example through Groovy!
- Again Groovy's multimethod will produce different result

```
3
2
3
3
```

It is odd?

```
public static void main(String[] args)
{
    // What's the output?
    System.out.println(isOdd(3));
    System.out.println(isOdd(-3));
}

private static boolean isOdd(int i)
{
    return i % 2 == 1;
}
```

It is odd?

```
true
false
```

- % 2 for -ve numbers does not work as you'd expect
- instead ask if it is even and negate

```
private static boolean isOdd(int i)
{
    return !(i % 2 == 0);
}
```

Computation

```
// What's the output?
System.out.println(2.0 - 1.1);
```

Computation

* Which language?!
* Java reports 0.8999999999999999
* Groovy reports 0.9
* float and double don't give you as much accuracy as BigDecimal
* Groovy uses that by default
* In Java, use BigDecimal for accuracy
* Note use of "" instead of new BigDecimal(2.0)
* Never use BigDecimal(double) constructor–exact double representation, so in accurate

```
System.out.println(
        new BigDecimal("2.0").subtract(new BigDecimal("1.1")));
```

Simple Math?

```
// What's the output?
int val = 1000000;
System.out.println(val * val / val);
```

Simple Math?

- Initial value fits in size of int
- Final result fits
- But intermediate values don't `-727`
- Make type large enough to hold intermediate results

```
int val = 1000000;
System.out.println((long)val * val / val);
```

Simple Addition

```java
// What's the output
double val = 100000000000000.00;
System.out.println(val);
val += 0.001;
System.out.println(val);
```

Simple Addition

- Magnitude of number is too large

```
1.0E14
1.0E14
```

- As magnitude increases, next representable nearest number if farther away
- You can find next nearest number using ulp (unit in the last place)

```
System.out.println(Math.ulp(1.0));
System.out.println(Math.ulp(1000.0));
System.out.println(Math.ulp(1000000.0));
System.out.println(Math.ulp(1000000000.0));
System.out.println(Math.ulp(1000000000000.0));
System.out.println(Math.ulp(1000000000000000.0));
```

```
2.220446049250313E-16
1.1368683772161603E-13
1.1641532182693481E-10
1.1920928955078125E-7
1.220703125E-4
0.125
```

References

* "Effective Java: Programming Languages Guide," Joshua Block, Addison Wesley, 2001.
* "Java Puzzlers: Traps, Pitfalls, and Corner Cases," Joshua Block and Neal Gafter, Addison Wesley, 2005.
* "Java Concurrency in Practice," Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea, Addison Wesley, 2006.
* "Programming Groovy: Dynamic Productivity for the Java Developer," Venkat Subramaniam, Pragmatic Bookshelf, 2008.

You can download examples and slides from

http://www.agiledeveloper.com - download

Thank You!

Please fill in your session evaluations

You can download examples and slides from

http://www.agiledeveloper.com - download