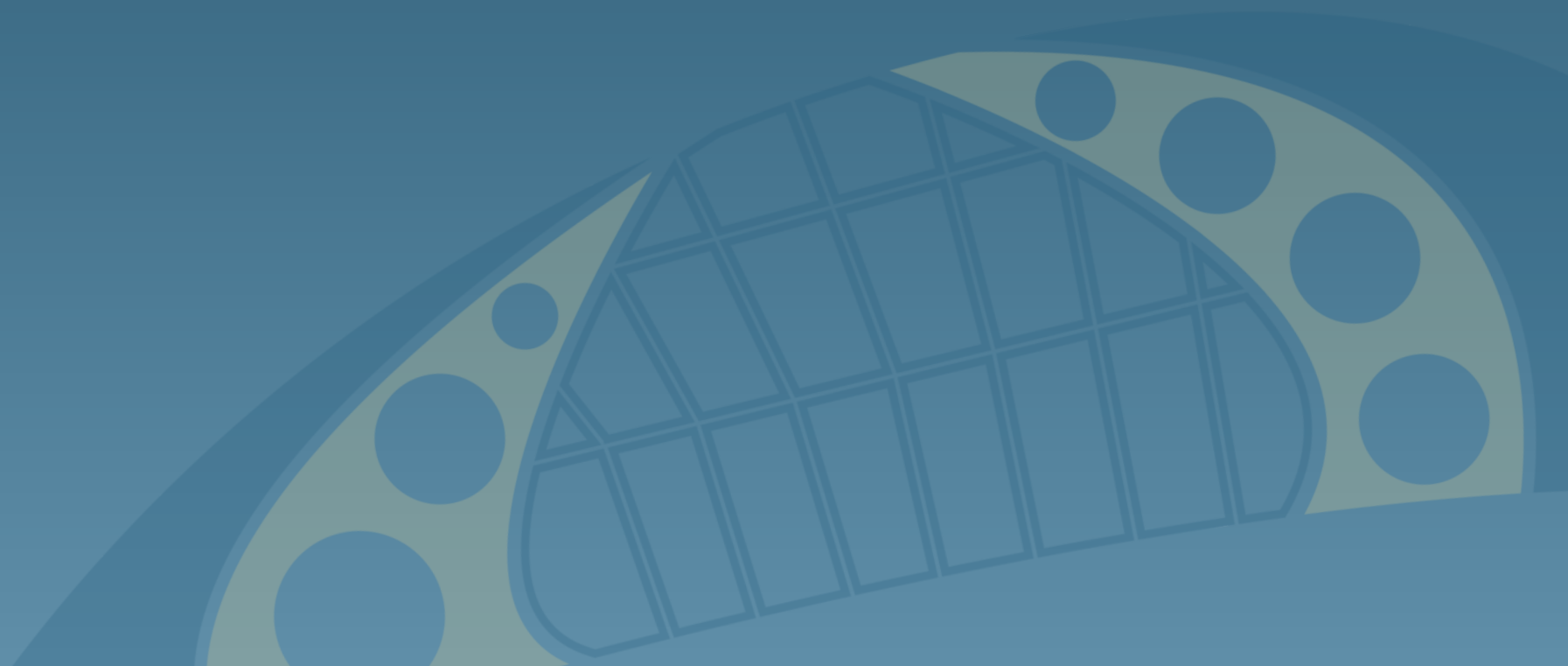


GENERIC IN JAVA

Computer Science II

CS 030

Donald J. Patterson



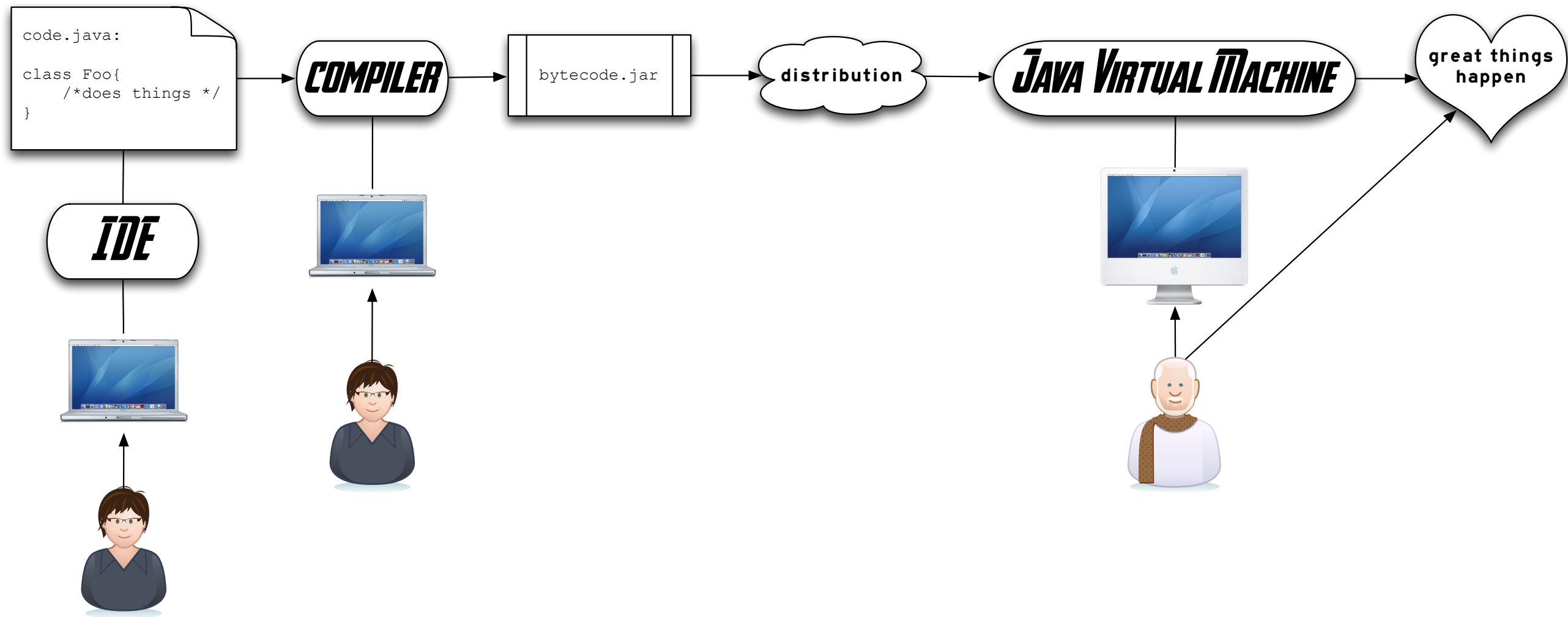
GENERICIS FROM 10,000 FEET

- **Generics** enable **types** (classes and interfaces) to be parameterized.
- The input to Generics are **types**
- The output of Generics are new **types**
- They are like parameters in methods
 - **but** they are about types, not data, not values
 - they are meta-data
- They let you re-use the same code with different inputs.

GENERICIS FROM 10,000 FEET

- Benefits:
 - Write less code (code re-use)
 - They enable programmers to implement generic algorithms once (less bugs)
- Stronger type checks at compile time.
- More errors are found at compile-time through static checking

GENERICIS FROM 10,000 FEET



GENERICs FROM 10,000 FEET

- Elimination of casts.
- The following code snippet without generics requires casting:

```
List list = new ArrayList();  
list.add("hello");  
String s = (String) list.get(0);
```

- When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();  
list.add("hello");  
String s = list.get(0);    // no cast
```

GENERICS FROM 10,000 FEET

```
public static void goodCode(String[] args) {  
    Vector longWords = new Vector();  
    int i;  
    for (i = 0; i < args.length; i++) {  
        if (args[i].length() > 4) {  
            longWords.add(args[i]);  
        }  
    }  
  
    for (i = 0; i < longWords.size(); i++) {  
        String word = (String) longWords.get(i);  
        System.out.println(word + ", length " + word.length());  
    }  
}
```

```
public static void badCode(String[] args) {  
    Vector longWords = new Vector();  
    int i;  
    for (i = 0; i < args.length; i++) {  
        if (args[i].length() > 4) {  
            longWords.add(args);  
        }  
    }  
  
    for (i = 0; i < longWords.size(); i++) {  
        String word = (String) longWords.get(i);  
        System.out.println(word + ", length " + word.length());  
    }  
}
```

- Questions:
 - What does the code on the left do?
 - What is different about the code on the right?
 - Is the code syntactically correct?

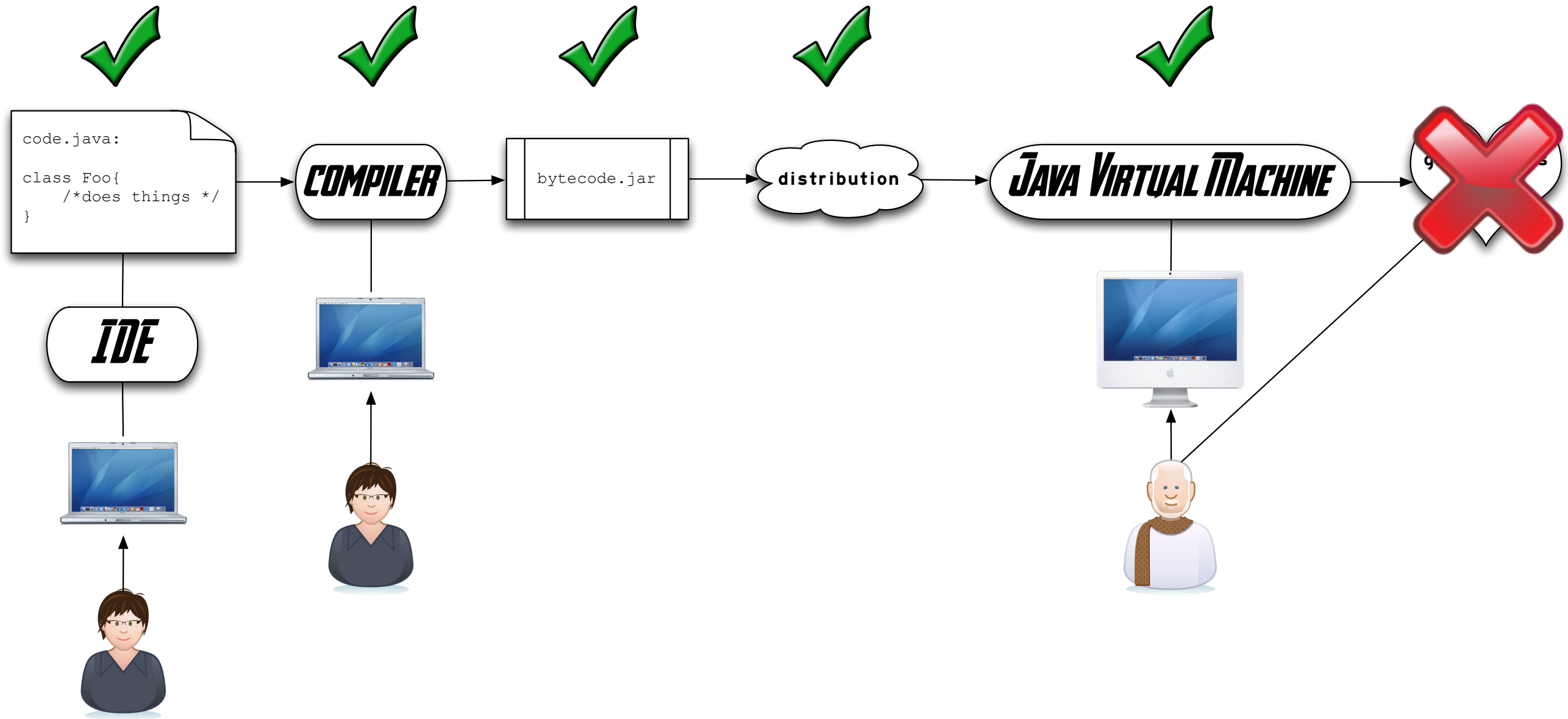
GENERICIS FROM 10,000 FEET

```
public static void goodCode(String[] args) {  
    Vector longWords = new Vector();  
    int i;  
    for (i = 0; i < args.length; i++) {  
        if (args[i].length() > 4) {  
            longWords.add(args[i]);  
        }  
    }  
  
    for (i = 0; i < longWords.size(); i++) {  
        String word = (String) longWords.get(i);  
        System.out.println(word + ", length " + word.length());  
    }  
}
```

```
public static void badCode(String[] args) {  
    Vector longWords = new Vector();  
    int i;  
    for (i = 0; i < args.length; i++) {  
        if (args[i].length() > 4) {  
            longWords.add(args);  
        }  
    }  
  
    for (i = 0; i < longWords.size(); i++) {  
        String word = (String) longWords.get(i);  
        System.out.println(word + ", length " + word.length());  
    }  
}
```

- Questions:
 - What does the code on the left do?
 - What is different about the code on the right?
 - Is the code syntactically correct?
- Let's try it

GENERICIS FROM 10,000 FEET



GENERICIS FROM 10,000 FEET

```
public static void goodCode(String[] args) {
    Vector longWords = new Vector();
    int i;
    for (i = 0; i < args.length; i++) {
        if (args[i].length() > 4) {
            longWords.add(args[i]);
        }
    }

    for (i = 0; i < longWords.size(); i++) {
        String word = (String) longWords.get(i);
        System.out.println(word + ", length " + word.length());
    }
}
```

```
public static void goodCodeWithGenerics(String[] args) {
    Vector<String> longWords = new Vector<String>();
    int i;
    for (i = 0; i < args.length; i++) {
        if (args[i].length() > 4) {
            longWords.add(args[i]);
        }
    }

    for (i = 0; i < longWords.size(); i++) {
        String word = longWords.get(i);
        System.out.println(word + ", length " + word.length());
    }
}
```

- Questions:
 - What is different about the code with Generics?

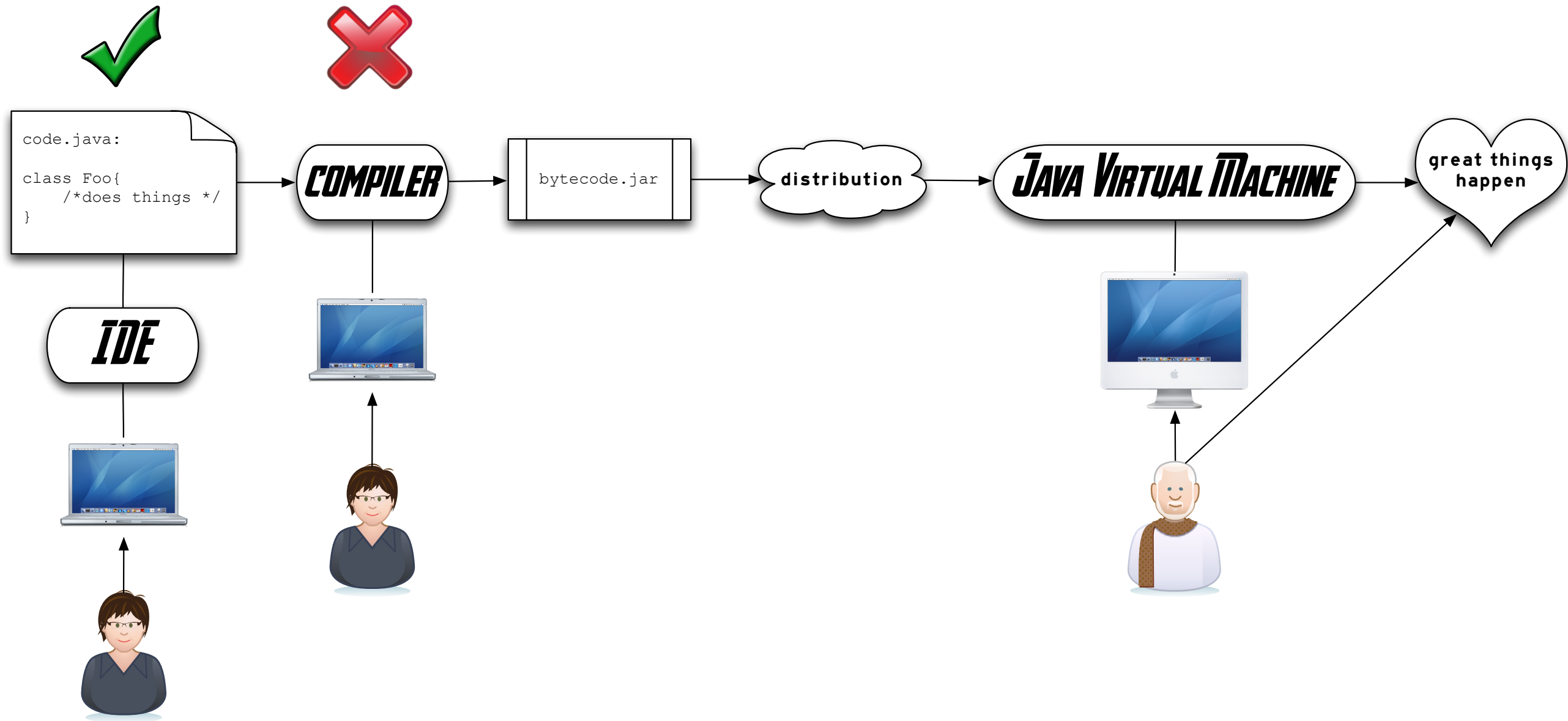
GENERICIS FROM 10,000 FEET

```
public static void goodCodeWithGenerics(String[] args) {  
    Vector<String> longWords = new Vector<String>();  
    int i;  
    for (i = 0; i < args.length; i++) {  
        if (args[i].length() > 4) {  
            longWords.add(args[i]);  
        }  
    }  
  
    for (i = 0; i < longWords.size(); i++) {  
        String word = longWords.get(i);  
        System.out.println(word + ", length " + word.length());  
    }  
}
```

```
public static void badCodeWithGenerics(String[] args) {  
    Vector<String> longWords = new Vector<String>();  
    int i;  
  
    for (i = 0; i < args.length; i++) {  
        if (args[i].length() > 4) {  
            longWords.add(args);  
        }  
    }  
  
    for (i = 0; i < longWords.size(); i++) {  
        String word = longWords.get(i);  
        System.out.println(word + ", length " + word.length());  
    }  
}
```

- Questions:
 - What happened to the code on the right?

GENERICIS FROM 10,000 FEET



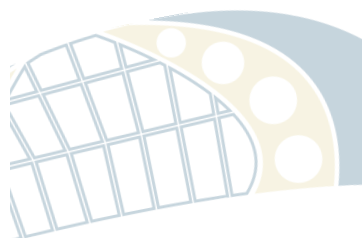
GENERICS FROM 10,000 FEET

```
public static void goodCodeWithGenerics(String[] args) {  
    Vector<String> longWords = new Vector<String>();  
    int i;  
    for (i = 0; i < args.length; i++) {  
        if (args[i].length() > 4) {  
            longWords.add(args[i]);  
        }  
    }  
  
    for (i = 0; i < longWords.size(); i++) {  
        String word = longWords.get(i);  
        System.out.println(word + ", length " + word.length());  
    }  
}
```

```
public static void goodCodeWithGenerics2(Identity[] args) {  
    Vector<Identity> tallPeople = new Vector<Identity>();  
    int i;  
    for (i = 0; i < args.length; i++) {  
        if (args[i].getHeight() > 6) {  
            tallPeople.add(args[i]);  
        }  
    }  
  
    for (i = 0; i < tallPeople.size(); i++) {  
        Identity person = tallPeople.get(i);  
        System.out.println(person.getName() + ", height " + person.getHeight());  
    }  
}
```

- Here's what's cool:
 - When the developer made the Vector class they had no idea that I was going to use it with Strings
 - I could have used it with some other class


```
public class Identity {  
  
    private String storedName;  
    private String storedPassword; /*for teaching don't ever actually do this */  
    private Integer storedHeight;  
  
    Identity(String name, String password){  
        storedName = name;  
        storedPassword = password;  
    }  
  
    public String getName() {  
        return storedName;  
    }  
  
    public void setName(String name) {  
        storedName = name;  
    }  
  
    private String getPassword() {  
        return storedPassword;  
    }  
  
    public void setHeight(Integer height){  
        storedHeight = height;  
    }  
  
    public Integer getHeight() {  
        return storedHeight;  
    }  
  
    public boolean setPassword(String oldPassword, String password) {  
        if(getPassword().equals(oldPassword)){  
            storedPassword = password;  
            return true;  
        }  
        return false;  
    }  
}
```



```
public boolean setPassword(String oldPassword, String password)
```

- Components of a method signature
 - visibility
 - return type
 - name
 - full name would include the class (and the package)
 - Identity.setPassword
- parameter list
 - parameter type
 - parameter name

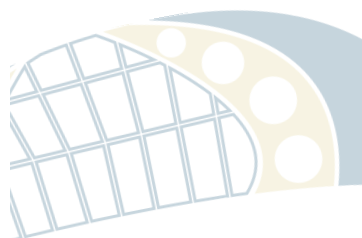


```
public boolean setPassword(String oldPassword, String password)
```

- Generics are a small language within a language for declaring types
- Let's work through one
- We are clever hackers and we want to let people have first and last names



```
public class Identity {  
  
    private String storedName;  
    private String storedPassword; /*for teaching don't ever actually do this */  
    private Integer storedHeight;  
  
    Identity(String name, String password){  
        storedName = name;  
        storedPassword = password;  
    }  
  
    public String getName() {  
        return storedName;  
    }  
  
    public void setName(String name) {  
        storedName = name;  
    }  
  
    private String getPassword() {  
        return storedPassword;  
    }  
  
    public void setHeight(Integer height){  
        storedHeight = height;  
    }  
  
    public Integer getHeight() {  
        return storedHeight;  
    }  
  
    public boolean setPassword(String oldPassword, String password) {  
        if(getPassword().equals(oldPassword)){  
            storedPassword = password;  
            return true;  
        }  
        return false;  
    }  
}
```



- But we also know that we can use this for more than just names
- Let's abstract it into a general Pair



```
public class Pair {  
  
    String storedFirst;  
    String storedSecond;  
  
    Pair(String first, String second){  
        storedFirst = first;  
        storedSecond = second;  
    }  
  
    public String getStoredFirst() {  
        return storedFirst;  
    }  
  
    public void setStoredFirst(String storedFirst) {  
        this.storedFirst = storedFirst;  
    }  
  
    public String getStoredSecond() {  
        return storedSecond;  
    }  
  
    public void setStoredSecond(String storedSecond) {  
        this.storedSecond = storedSecond;  
    }  
  
}
```



```

public class Pair {

    String storedFirst;
    String storedSecond;

    Pair(String first, String second){
        storedFirst = first;
        storedSecond = second;
    }

    public String getStoredFirst() {
        return storedFirst;
    }

    public void setStoredFirst(String storedFirst) {
        this.storedFirst = storedFirst;
    }

    public String getStoredSecond() {
        return storedSecond;
    }

    public void setStoredSecond(String storedSecond) {
        this.storedSecond = storedSecond;
    }

}

```

```

public class IdentityName {

    private Pair storedName;
    private String storedPassword; /*for teaching don't ever actually do this */
    private Integer storedHeight;

    IdentityName(String firstName, String lastName, String password){
        storedName = new Pair(firstName, lastName);
        storedPassword = password;
    }

    public String getName() {
        return storedName.getStoredFirst() + " " + storedName.getStoredSecond();
    }

    public void setName(Pair name) {
        storedName = name;
    }

    private String getPassword() {
        return storedPassword;
    }

    public void setHeight(Integer height){
        storedHeight = height;
    }

    public Integer getHeight() {
        return storedHeight;
    }

    public boolean setPassword(String oldPassword, String password) {
        if(getPassword().equals(oldPassword)){
            storedPassword = password;
            return true;
        }
        return false;
    }

}

```



- Mission Accomplished!
- Word has spread far and wide and now we've been hired by an EMR company
- They want us to add BMI to our Identity class
 - That's a height and weight combo
 - Hey! That sounds like another Pair!
 - We've already done that right?




```

public class IdentityName {

    private Pair storedName;
    private String storedPassword; /*for teaching don't ever actually do this */
    private Pair storedBMI;

    IdentityName(String firstName, String lastName, String password, Double height, Double weight){
        storedName = new Pair(firstName, lastName);
        storedPassword = password;
        storedBMI = new Pair(height, weight);
    }

    public String getName() {
        return storedName.getStoredFirst() + " " + storedName.getStoredSecond();
    }

    public void setName(Pair name) {
        storedName = name;
    }

    private String getPassword() {
        return storedPassword;
    }

    public void setBMI(Pair bmi){
        storedBMI = bmi;
    }

    public Double getBMI() {
        return (storedBMI.getStoredSecond()*703.0)/(storedBMI.getStoredFirst()*storedBMI.getStoredFirst());
    }

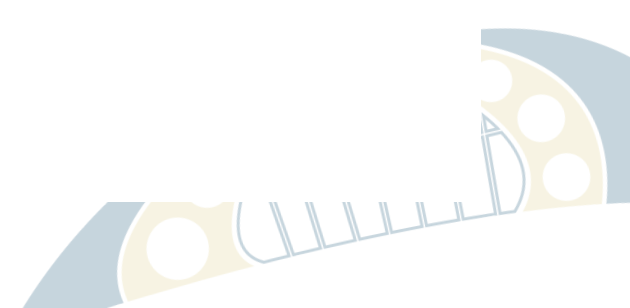
    public boolean setPassword(String oldPassword, String password) {
        if(getPassword().equals(oldPassword)){
            storedPassword = password;
            return true;
        }
        return false;
    }

}

```

The constructor Pair(Double, Double) is undefined.

Multiple markers at this line
 - The operator * is undefined for the argument type(s) java.lang.String, java.lang.String
 - The operator * is undefined for the argument type(s) String, double



- Hmm...
- That's easy enough to fix. Let's just make another Pair class...



- Conceptually our Pair class is agnostic to what kind of type gets used with it
 - it doesn't really matter
 - But the Java language is tying our hands
- But we are stuck having to define a new class for every single application that needs a different type
- Enter Generics
- Generics let you manipulate parameters without knowing their type, but without losing static type checking in the compiler



```
public class PairGeneric<F,S> {  
    F storedFirst;  
    S storedSecond;  
  
    PairGeneric(F first, S second){  
        storedFirst = first;  
        storedSecond = second;  
    }  
  
    public F getStoredFirst() {  
        return storedFirst;  
    }  
  
    public void setStoredFirst(F storedFirst) {  
        this.storedFirst = storedFirst;  
    }  
  
    public S getStoredSecond() {  
        return storedSecond;  
    }  
  
    public void setStoredSecond(S storedSecond) {  
        this.storedSecond = storedSecond;  
    }  
}
```



```
public class IdentityGenerics {

    private PairGeneric<String,String> storedName;
    private String storedPassword; /*for teaching don't ever actually do this */
    private PairGeneric<Double,Double> storedBMI;

    IdentityGenerics(String firstName, String lastName, String password, Double height, Double weight){
        storedName = new PairGeneric<String,String>(firstName, lastName);
        storedPassword = password;
        storedBMI = new PairGeneric<Double,Double>(height, weight);
    }

    public String getName() {
        return storedName.getStoredFirst() + " " + storedName.getStoredSecond();
    }

    public void setName(PairGeneric<String,String> name) {
        storedName = name;
    }

    private String getPassword() {
        return storedPassword;
    }

    public void setBMI(PairGeneric<Double,Double> bmi){
        storedBMI = bmi;
    }

    public Double getBMI() {
        return (storedBMI.getStoredSecond()*703.0)/(storedBMI.getStoredFirst()*storedBMI.getStoredFirst());
    }

    public boolean setPassword(String oldPassword, String password) {
        if(getPassword().equals(oldPassword)){
            storedPassword = password;
            return true;
        }
        return false;
    }
}
```

- Now, mission accomplished
- That's 80% of everything to know about Generics
- It's basically that easy, but there are some details that you need to know
- The final 20%....



Generics and primitive types don't play well together



- You can't plug in a primitive type to a Generic



```
int foo = 1;  
int bar = 2;  
new PairGeneric<int,int>(foo,bar);
```

- You can't use a Generic in an Array (it's primitive)



```
Pair[] foo = new Pair[100];  
PairGeneric<Double,Double>[] bar = new PairGeneric<Double,Double>[100];
```



Sometimes you care just a little bit about the type

- When writing a Generic you actually have a little more control over how a developer uses it

```
public class PairSortaGeneric<F extends Number, S extends PairDoubleDouble> {  
  
    F storedFirst;  
    S storedSecond;  
  
    PairSortaGeneric(F first, S second){  
        storedFirst = first;  
        storedSecond = second;  
    }  
  
    public F getStoredFirst() {  
        return storedFirst;  
    }  
  
    public void setStoredFirst(F storedFirst) {  
        this.storedFirst = storedFirst;  
    }  
  
    public S getStoredSecond() {  
        return storedSecond;  
    }  
  
    public void setStoredSecond(S storedSecond) {  
        this.storedSecond = storedSecond;  
    }  
  
    public Number scaleIt(){  
        return storedFirst.doubleValue() * storedSecond.getStoredFirst();  
    }  
}
```



Generics can be extended

- How could you make a Quad?

```
public class QuadGeneric<S,T,U,V> {  
    PairGeneric<S,T> foo;  
    PairGeneric<U,V> bar;  
  
    QuadGeneric(S s,T t,U u,V v){  
        foo = new PairGeneric<S,T>(s,t);  
        bar = new PairGeneric<U,V>(u,v);  
    }  
  
    S getFirst(){  
        return foo.getStoredFirst();  
    }  
  
    T getSecond(){  
        return foo.getStoredSecond();  
    }  
  
    U getThird(){  
        return bar.getStoredFirst();  
    }  
  
    V getFourth(){  
        return bar.getStoredSecond();  
    }  
}
```

```
public class QuadGeneric<S,T,U,V> extends PairGeneric<S, T> {  
    PairGeneric<U,V> foo;  
  
    QuadGeneric(S s,T t,U u,V v){  
        super(s,t);  
        foo = new PairGeneric<U,V>(u,v);  
    }  
  
    S getFirst(){  
        return getStoredFirst();  
    }  
  
    T getSecond(){  
        return getStoredSecond();  
    }  
  
    U getThird(){  
        return foo.getStoredFirst();  
    }  
  
    V getFourth(){  
        return foo.getStoredSecond();  
    }  
}
```

Generic Types are Atomic

```
Object someObject = new Object();  
Integer someInteger = new Integer(10);  
someObject = someInteger;    // OK
```

```
public void someMethod(Number n) { /* ... */ }  
  
someMethod(new Integer(10));    // OK  
someMethod(new Double(10.1));  // OK
```

```
Integer foo = new Integer(10);  
Double bar = new Double(10.1);  
PairGeneric<Number,Number> p= new PairGeneric<Number,Number>(foo,bar);
```

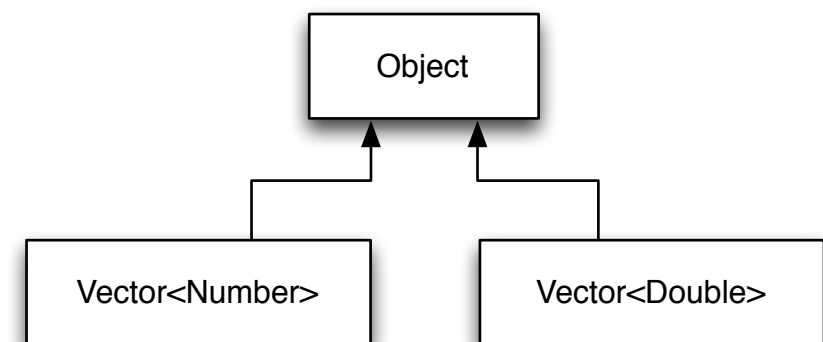
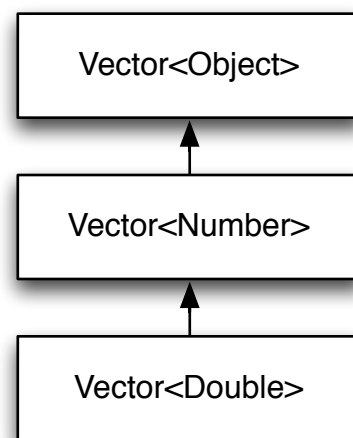
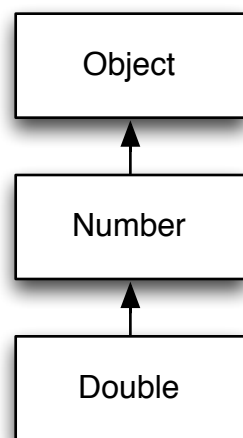
- You cannot inherit just a portion of a type

Generic Types are Atomic

- You cannot inherit just a portion of a type

```
public Double sum(Vector<Number> v) {  
    double running = 0.0;  
    for(int i = 0 ; i < v.size(); i++){  
        running += v.get(i).doubleValue();  
    }  
    return running;  
}
```

```
Vector<Double> bar = new Vector<Double>();  
bar.add(10.1);  
bar.add(20.2);  
Double result = sum(bar);
```



Work it out

- Will the following code compile?

```
public class Algorithm{  
    public T max(T x, T y) {  
        return x > y ? x : y;  
    }  
}
```



Generics from 10,000 feet

- **Generics** enable **types** (classes and interfaces) to be **parameterized**.
- The input to Generics are **types**
- The output of Generics are new **types**
- They are like parameters in methods
 - **but** they are about types, not data, not values
 - they are meta-data
- They let you re-use the same code with different inputs.

Generics from 10,000 feet

- Benefits:
 - Write less code (code re-use)
 - Enabling programmers to implement generic algorithms.
- Stronger type checks at compile time.
- More errors are found at compile-time through static checking



WESTMONT **INSPIRED**
— COMPUTING LAB —