One big problem with “reusing” code such as we have written is that we have to create an ADT (e.g., stack or queue) for a specific type:

```java
class ArrayIntStack {
    private int[] A = new int[8];
    private int next = 0;

    public void push(int key) {
        A[next] = key;
        ++next;
    }

    public int pop() {
        int temp = A[next-1];
        --next;
        return temp;
    }

    public boolean isEmpty() {
        return (next == 0);
    }

    public int size() {
        return next;
    }
}
```
Polymorphic ADTs

If we need a stack for a different type of data, we have to create another whole class, although we can do this easily by using the editor to replace int by, e.g., double:

class ArrayIntStack { 
    private int [] A = new int[8];
    private int next = 0;

    public void push(int key) { 
        A[next] = key;
        ++next;
    }

    public int pop() { 
        int temp = A[next-1];
        --next;
        return temp;
    }

    public boolean isEmpty() { 
        return (next == 0);
    }

    public int size() { 
        return next;
    }
}

class ArrayDoubleStack { 
    private double [] A = new double[8];
    private int next = 0;

    public void push(double key) { 
        A[next] = key;
        ++next;
    }

    public double pop() { 
        double temp = A[next-1];
        --next;
        return temp;
    }

    public boolean isEmpty() { 
        return (next == 0);
    }

    public int size() { 
        return next;
    }
}
But this is a really terrible idea: not only might we make a mistake and change something that should be an int into a double, we now have **multiple versions** of the same code that have to be **stored** and **maintained**. Or more than two! If we change one, we have to change the others! It’s a massive version of the “Multiple Update Problem.”

```java
class ArrayIntStack {
    private int[] A = new int[8];
    private int next = 0;

    public void push(int key) {
        A[next] = key;
        ++next;
    }

    public int pop() {
        int temp = A[next-1];
        --next;
        return temp;
    }

    public boolean isEmpty() {
        return (next == 0);
    }

    public int size() {
        return next;
    }
}

class ArrayDoubleStack {
    private double[] A = new double[8];
    private int next = 0;

    public void push(double key) {
        A[next] = key;
        ++next;
    }

    public double pop() {
        double temp = A[next-1];
        --next;
        return temp;
    }

    public boolean isEmpty() {
        return (next == 0);
    }

    public int size() {
        return next;
    }
}

class ArrayStringStack {
    private String[] A = new String[8];
    private int next = 0;

    public void push(String key) {
        A[next] = key;
        ++next;
    }

    public String pop() {
        String temp = A[next-1];
        --next;
        return temp;
    }

    public boolean isEmpty() {
        return (next == 0);
    }

    public int size() {
        return next;
    }
}

class ArrayCharStack {
    private char[] A = new char[8];
    private int next = 0;

    public void push(char key) {
        A[next] = key;
        ++next;
    }

    public char pop() {
        char temp = A[next-1];
        --next;
        return temp;
    }

    public boolean isEmpty() {
        return (next == 0);
    }

    public int size() {
        return next;
    }
}
```
**Polymorphic ADTs**

**Interesting question:** is there anything essential about the data and methods for a Stack that really depends on the type of the elements stored?

class ArrayIntStack {
    private int[] A = new int[8];
    private int next = 0;

    public void push(int key) {
        A[next] = key;
        ++next;
    }

    public int pop() {
        int temp = A[next-1];
        --next;
        return temp;
    }

    public boolean isEmpty() {
        return (next == 0);
    }

    public int size() {
        return next;
    }
}

class ArrayDoubleStack {
    private double[] A = new double[8];
    private int next = 0;

    public void push(double key) {
        A[next] = key;
        ++next;
    }

    public double pop() {
        double temp = A[next-1];
        --next;
        return temp;
    }

    public boolean isEmpty() {
        return (next == 0);
    }

    public int size() {
        return next;
    }
}
Polymorphic ADTs

Interesting question: is there anything essential about the data and methods for a Stack that really depends on the type of the elements stored?

NO! You want proof? How about this: we made the change using Find/Replace.....

class ArrayIntStack {
    private int [] A = new int[8];
    private int next = 0;

    public void push(int key) {
        A[next] = key;
        ++next;
    }

    public int pop() {
        int temp = A[next-1];
        --next;
        return temp;
    }

    public boolean isEmpty() {
        return (next == 0);
    }

    public int size() {
        return next;
    }
}

class ArrayDoubleStack {
    private double [] A = new double[8];
    private int next = 0;

    public void push(double key) {
        A[next] = key;
        ++next;
    }

    public double pop() {
        double temp = A[next-1];
        --next;
        return temp;
    }

    public boolean isEmpty() {
        return (next == 0);
    }

    public int size() {
        return next;
    }
}
Java Generic Types

Java’s Generic Type system solves this problem.

The idea is to parameterize a class as if it were a method.

Consider the following (silly) methods:

```java
void repeat2(string s) {
    System.out.println(s); System.out.println(s);
}

void repeat3(string s) {
    System.out.println(s); System.out.println(s);
    System.out.println(s);
}

// to use:

repeat3("I am sick of cold weather");
```
Java Generic Types

To parameterize something, we take an essential part out and make it a variable that we can instantiate; then the same code can be used in a variety of ways. It avoids the multiple update problem and makes it possible to reuse code later.

Obviously, instead of writing repeat2(), repeat3(), …., repeat100000()….. we should just make the number of repetitions a parameter:

```java
void repeat (string s, int N) {
    for(int I = 0; i < N; ++i)
        System.out.println(s);
}
```

// to use:

Repeat(“I am sick of cold weather”, 2);

Recall: Parameter passing is an assignment:

N = 2;
Java Generic Types

The EXACT SAME THING can be done with a Java class: we can take out one or more type names and make it a parameter. The only thing to get straight is the syntax, which is a little different than parameters in methods. Here is the original (simplified) code for a Stack holding ints, with the places where the type int is used to refer to the type stored in the stack:

```java
public class Stack {
    private class Node {
        int item;
        Node next;
        public Stack(int n, Node p) {
            item = n; next = p;
        }
    }
    private Node top = null;

    public void push(int n) {
        top = new Node(n, top);
    }

    public int pop() {
        int temp = top.item; top = top.next; return temp;
    }
}
```
Now we replace the type name String with a variable (can be any name but usually it is called Item or T or something descriptive, and capitalized):

```java
public class Stack {
    private class Node {
        Item item;
        Node next;
        public Stack(Item n, Node p) {
            item = n; next = p;
        }
    }
    private Node top = null;

    public void push(Item n) {
        top = new Node(n, top);
    }

    public Item pop() {
        Item temp = top.item; top = top.next; return temp;
    }
}
```
Java Generic Types

Now we replace the type name int with a variable (can be any name but usually it is called Item or T or something descriptive, and capitalized);

Next we add the parameter list to the name of the class in angle brackets: < >

```java
public class Stack<Item> {
    private class Node {
        Item item;
        Node next;
        public Stack(Item n, Node p) {
            item = n; next = p;
        }
    }
    private Node top = null;

    public void push(Item n) {
        top = new Node(n, top);
    }
    public Item pop() {
        Item temp = top.item; top = top.next; return temp;
    }
}
```

Note that there is no type (as in “int N”) because Item is itself a type!
Java Generic Types

Now we replace the type name int with a variable (can be any name but usually it is called Item or T or something descriptive, and capitalized);

Next we add the parameter list to the name of the class in angle brackets: < >

To use the class, we must supply an actual parameter whenever we use the name Stack:

```java
public class Stack<Item> { 
    private class Node { 
        ... etc ...
    }
    private Node top = null;

    public void push(Item n) { 
        top = new Node(n, top);
    }
    public Item pop() { 
        Item temp = top.item; top = top.next; return temp;
    }
}

Stack<String> S = new Stack<String>();
```
Java Generic Types

Now we replace the type name int with a variable (can be any name but usually it is called Item or T or something descriptive, and capitalized);

Next we add the parameter list to the name of the class in angle brackets: < >

To use the class, we must supply an actual parameter whenever we use the name IntStack;

The effect of this is exactly as if we had written the code with the actual parameter Integer substituted for the formal parameter Item!

```java
public class Stack {
    private class Node {
        // etc ...
    }
    private Node top = null;

    public void push(String n) {
        top = new Node(n, top);
    }
    public String pop() {
        String temp = top.item; top = top.next; return temp;
    }
}
```
Java Generic Types

There are two more things you need to know:

First, Java doesn’t allow you to declare an array of generic types, and though there is an (ugly) hack to solve this, we recommend you don’t bother!

Second, you can ONLY use reference types (e.g. Strings and classes), and if you want to use primitive types, you have to use the Wrapper Types instead:

Integer = int    Double = double    Boolean = boolean    Character = char

There is not much difference between these, except that variables are now references.
Ok, ok, ONE more thing to know: There isn’t much you can do with generics, except move data around (stack, queue, …· um, not much else!).

What is MORE useful is to have a generic type with SOME properties, such as it can be ordered, or it has certain methods that are useful.

In Java, a generic type can be forced to satisfy an interface, so that at least you know SOMETHING about the type that instantiates the generic class.

**Generic Interfaces**

Interfaces specify what methods must be public in a class. Interfaces can be generic, just as classes can. The most common such interface is the generic Comparable interface, which is built in to the Java language:

```java
interface Comparable<Item> {
    int compareTo(Item);
}
```
Generic interfaces are just like normal interfaces, except you have to instantiate them with a class or wrapper type, and then when you implement the interface, you have to specify the types involved:

```java
public class GenericPQueue<Item extends Comparable<Item>> {
    // now you can assume that Item has a compareTo(Item) method
}
```

```java
interface Comparable<Item> {
    int compareTo(Item);
}
```