Generics

Objectives

- Topics
  - Using built-in generics
  - Writing generic classes
  - Wildcards for generics
  - Writing Enum classes

Generics

- A ***generic*** is a class (or interface or method) with a placeholder – like a parameter to that class required to be filled in later with some *actual* class (not a primitive type)
- The advantage of generic code is its reusability: you can substitute (almost) any class
  - Almost, because the methods you use on the generic must exist and make sense
- The downside is, generic code is harder to read and understand

Built-in Generics

- We've already used many built-in generics:
  - ArrayList<class name goes here> and all of its relatives
    - Example: ArrayList<String>
  - Comparable<class name goes here> and many other interfaces
    - Example: Comparable<Customer>
- You could, instead, type things as "Object", because every class is a child or later relative of Object
  - But doing this usually requires typecasting

Built-in Generics, cont.

- Also remember that a generic can take multiple class parameters, as in:

```
Map<String, ArrayList<Movie>>
Pair<Integer, String>
```

- It's not limited to two
  - although the more you have, the more confusing it is

Creating Generics

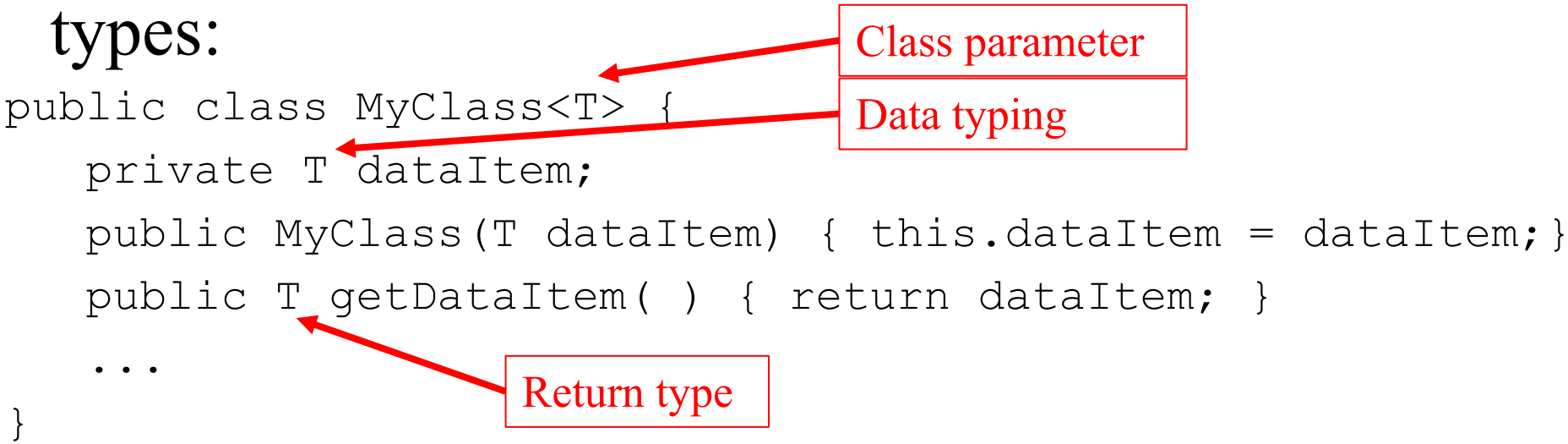
- To create your own generic class, the basic idea is to use <T> with the class name:

```
public class MyClass<T> { … }
```

- There's nothing special about "T", it's just the commonly used name for "template"
  - Although there are other standard usages:
    - K for key
    - N for number
    - E for element

Creating Generics, cont.

- Then use T as the type name for data, parameters, and return types:




```
public class MyClass<T> {
    private T dataItem;
    public MyClass(T dataItem) { this.dataItem = dataItem;}
    public T getDataItem( ) { return dataItem; }
    ...
}
```

- And using the class is just like built-ins:

```
MyClass<String> myclass = new MyClass<>( );
```

Generic Classes

- If the generic class calls any of T's methods …
  - You might be asking for trouble: if *any* class can be substituted for T, how do you know what methods it has?

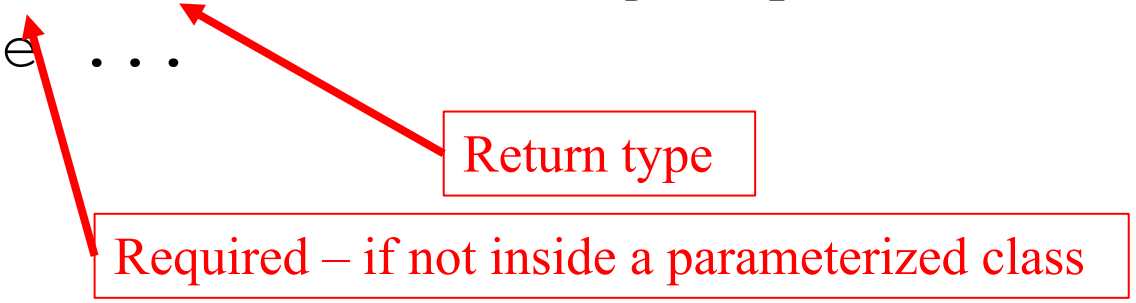
```
private T dataItem;
// Then is some method:
dataItem.doTheThing( ); // Does doTheThing exist?
```

  - That's why built-in classes often require that T implement some interface(s) so that there's a guarantee that some methods exist
  - But even then, it's iffy: `ArrayList` needs `Comparable`'s `compareTo()` to sort, but you can create an `ArrayList` without it.

Generic Methods

- A generic method uses generic parameters – these can be in any kind of class, not just generic classes

```
public static <T> int doTheThing(T parameter) {
    ... some code ...
}
```



- To call it, just pass the parameter(s)

```
int value = doTheThing("dog")
```

- But again, what methods can you use on the parameter?

Bounded Type Parameters

- Sometimes, you may want to limit the types that can be substituted for the generic – for example, to ensure that some methods exist
- Use the notation <T extends SomeClass> to restrict what gets substituted to children of SomeClass
  - Here, Note or any class that extends Note:

```
public class MyClass<T extends Note> { ...

// Then:
MyClass<TimedNote> obj = new MyClass<>();
```

Wild Cards

- But that doesn't work in this situation: parent `Shape`, child `Circle`

```
public class Canvas {
   public void drawAll(List<Shape> shapeList) {
      for (Shape s: shapeList) { s.draw(this); }

// Then in main:
List<Shape> slist = new ArrayList<>();
slist.add(new Circle());
Canvas c = new Canvas();
c.drawAll(slist);     // Okay

List<Circle> clist = new ArrayList<>();
clist.add(new Circle());
c.drawAll(clist);     // Compiler error
```

Wild Cards, cont.

- This is because List<Circle> is ***not*** a child of List<Shape>, even though Circle is a child of Shape
- So use a wild card instead: `<? extend SomeClass>`

```
public class Canvas {
   public void drawAll(List<? extends Shape> shapeList) {
      for (Shape s: shapeList) { s.draw(this); }


List<Circle> clist = new ArrayList<>();
clist.add(new Circle());
c.drawAll(clist);     // Okay
```