Object Oriented Programming and Design in Java

Session 14
Instructor: Bert Huang

Announcements

- Homework 3 out. Due **Monday**, Apr. 5th
- Midterm solutions and grades posted
- Office hour change

Sun	Mon	Tue	Wed	Thu	Fri
John 1-3	Class 11-12:15		Class 11-12:15 Bert 2-4 Yipeng 4-6		Lauren 11-1

Review

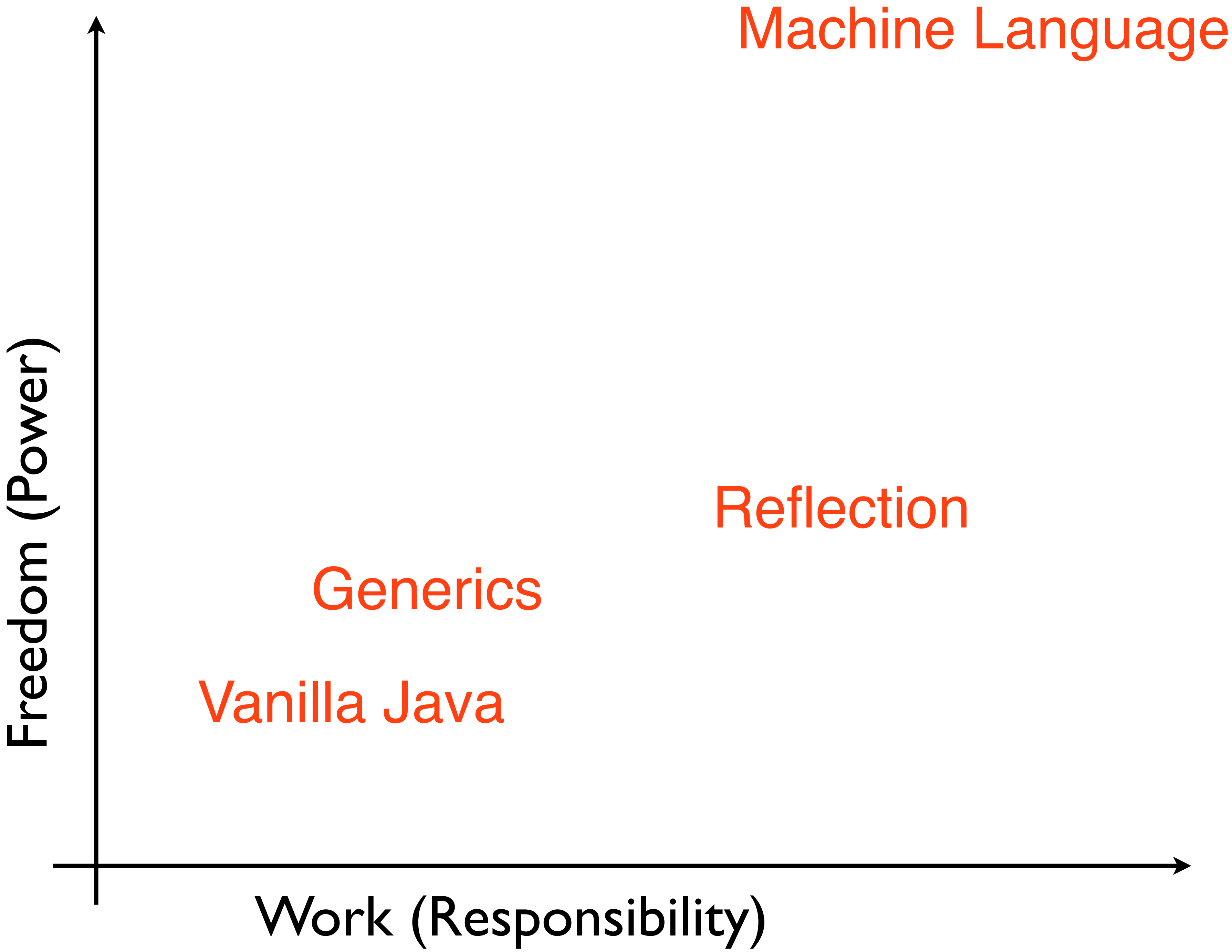
- Returned midterm
    - Statistics, common mistakes
- Cloneable
- Serializable
- Reflection
    - Class, Method, Field objects

Today’s Plan

- Generics
  - Generic types
  - Generic methods
  - Type bounds and wildcards
  - Type erasure

Tradeoffs

- With more powerful tools, more work is necessary to ensure programs are robust and stable
- We saw extreme power in reflection, in that it allows very general code
    - makes types less restrictive; we can handle them dynamically at runtime
- but code using reflection can be hard to maintain



Machine Language

Freedom (Power)

Reflection

Generics

Vanilla Java

Work (Responsibility)

Old-Fashioned Generics

-
  ```
  public class ArrayList {
      void add(Object obj) { ... }
      Object get(int index) { ... }
  }
  ```
- Any Object subclass works
- Runtime exception when typecasting fails
- We could use reflection to check all casts

Modern Java Generics

- Java since version 5 has allowed generic type placeholders
- Write general code, declaring generic classes and methods
- *Instantiate* with actual types for placeholders

Generics We've Used

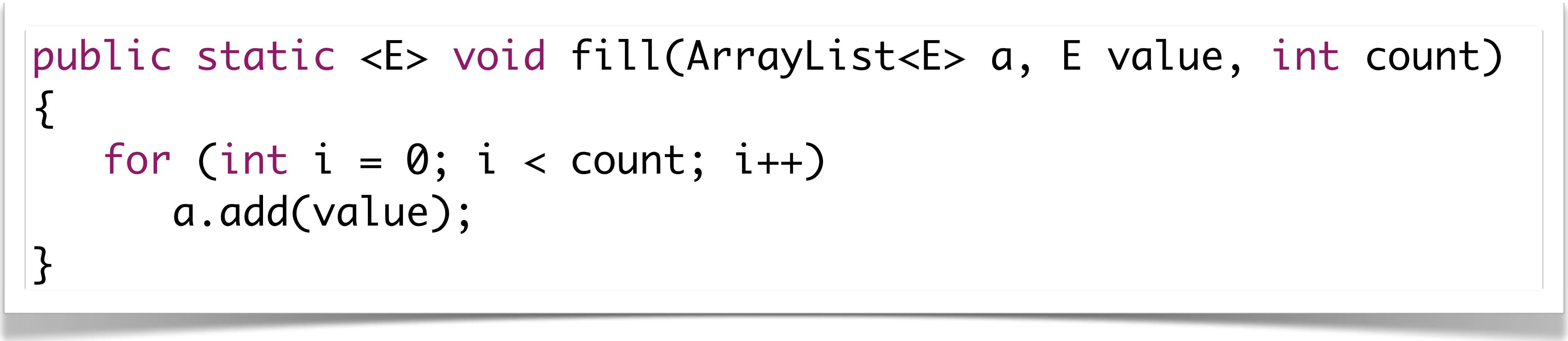
- We have used a few generic types from the standard library
- `ArrayList<T>` stores objects of type T
- `Iterable<T>` iterates over objects of type T
  - or implicitly with enhanced for loop
    `for (Shape s : Model)`

Generic Types

- Declared with a generic placeholder
- `public class Box<T> { ... }`
  - `Box<String> b = new Box<String>();`
  - `Box<Integer> b = new Box<Integer>();`
- `public class Pair<T,U> { ... }`
  - `Pair<String, Date> p = new Pair<String, Date>();`

Generic Methods

- We can use generic types in methods, which get resolved dynamically when the method is called

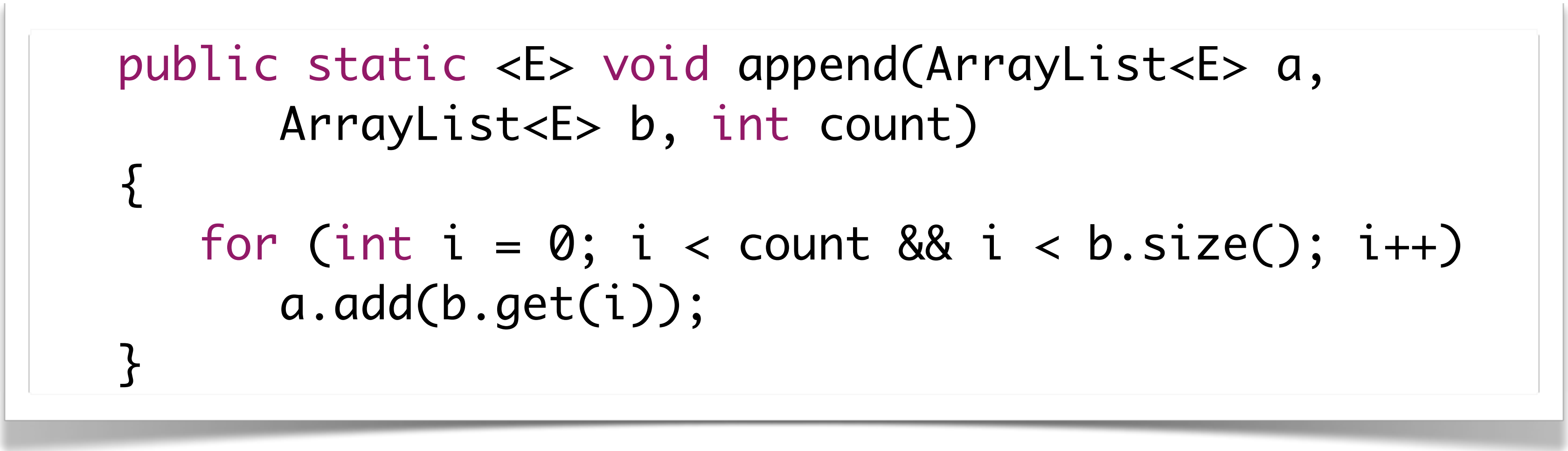


```
public static <E> void fill(ArrayList<E> a, E value, int count)
{
   for (int i = 0; i < count; i++)
      a.add(value);
}
```

- This checks that the ArrayList and value are of the appropriate type at compile time

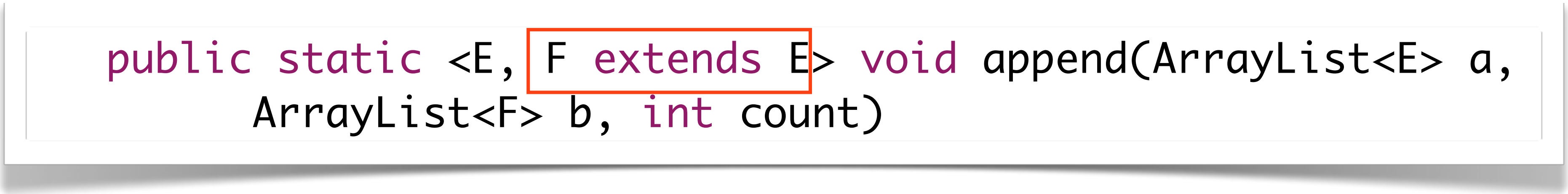
Type bounds

- Occasionally, generic types are too restrictive



```
public static <E> void append(ArrayList<E> a,
      ArrayList<E> b, int count)
{
   for (int i = 0; i < count && i < b.size(); i++)
      a.add(b.get(i));
}
```

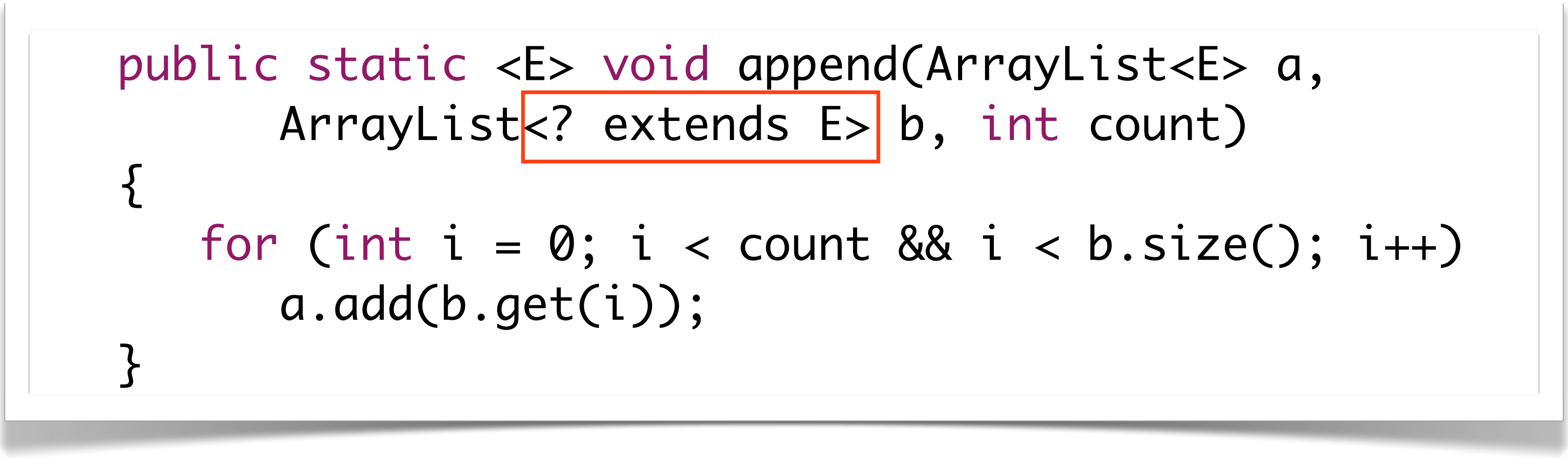
- We can use a *type bound* to relax restrictions



```
public static <E, F extends E> void append(ArrayList<E> a,
      ArrayList<F> b, int count)
```

Wildcards

- Type bounds still require that the client defines the generic types
- Sometimes this is undesirable, so we can use wildcards instead

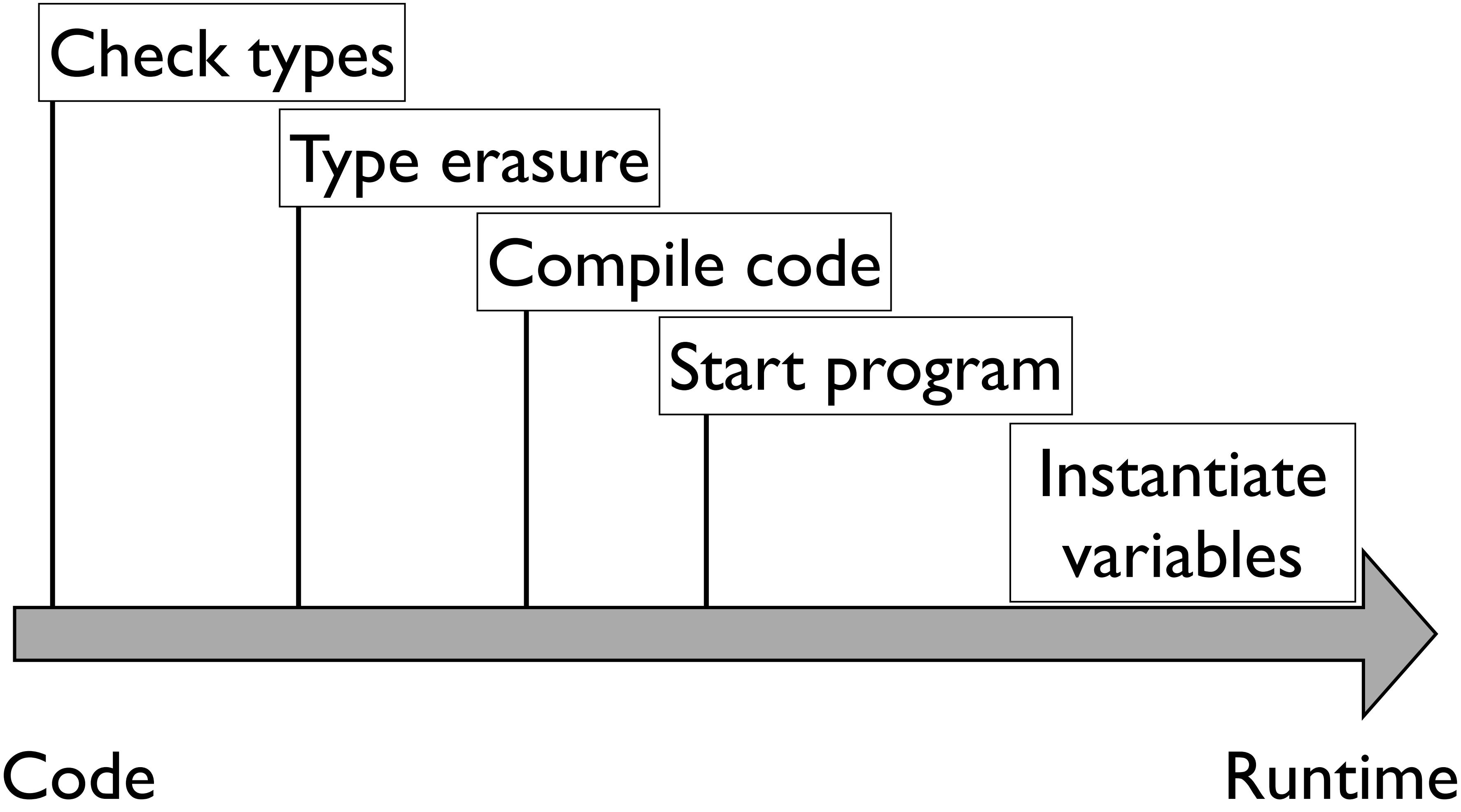


```
public static <E> void append(ArrayList<E> a,
      ArrayList<? extends E> b, int count)
{
   for (int i = 0; i < count && i < b.size(); i++)
      a.add(b.get(i));
}
```

Type Erasure

- After javac checks correct type usage with generics, it strips all types from the code into *raw types*
- The resulting code is similar to old-fashioned “generic” code, using Object variables (or the most general superclass)
- This allows compatibility with older code
  - but unfortunately leads to some limitations

Generics Compilation Process



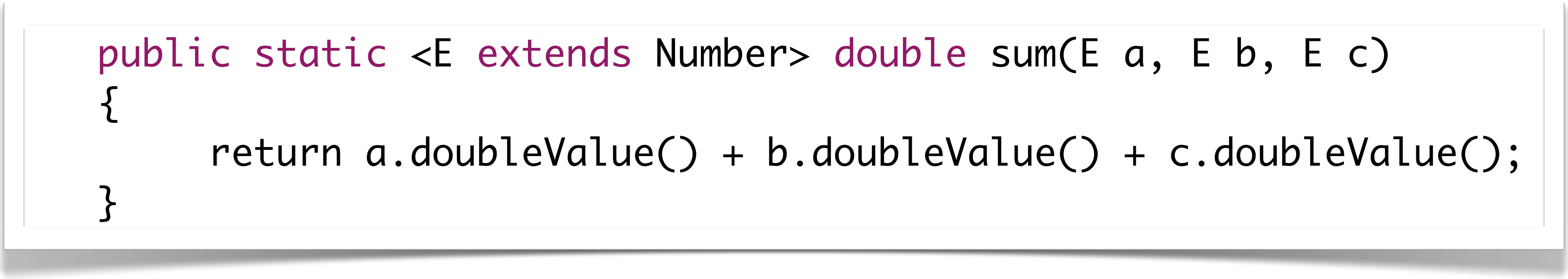
Erasure Example

```
public static <E> void fill(ArrayList<E> a, E value, int count)
{
   for (int i = 0; i < count; i++)
      a.add(value);
}
```

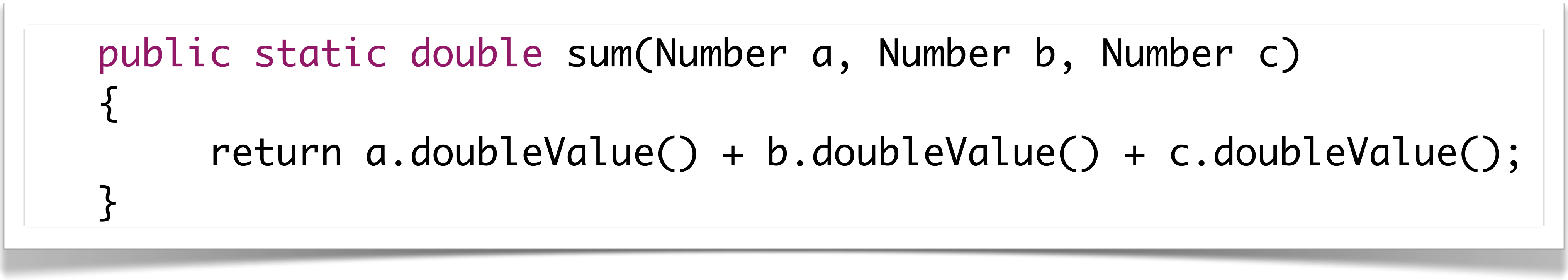
also type-erased

```
public static void fill(ArrayList a, Object value, int count)
{
   for (int i = 0; i < count; i++)
      a.add(value);
}
```

Erasure with Type Bounds



```
public static <E extends Number> double sum(E a, E b, E c)
{
     return a.doubleValue() + b.doubleValue() + c.doubleValue();
}
```



```
public static double sum(Number a, Number b, Number c)
{
     return a.doubleValue() + b.doubleValue() + c.doubleValue();
}
```

Erasure with Type Bounds 2

```
public static <E, F extends E> void append(ArrayList<E> a,
      ArrayList<F> b, int count)
{
   for (int i = 0; i < count && i < b.size(); i++)
      a.add(b.get(i));
}
```

OK because types are checked before type-erasure

```
public static void append(ArrayList a,
      ArrayList b, int count)
{
   for (int i = 0; i < count && i < b.size(); i++)
      a.add(b.get(i));
}
```

Compatibility Issues

- Generics in Java aren’t perfect
- One annoying problem comes because of type erasure:
  - We can’t create new objects of generic types

```
public <E> void addNew(ArrayList<E> a)
{
    a.add(new E());
}
```

becomes `new Object()`

Arrays of Generics

- Similarly, we cannot create an array of generics
- Type erasure doesn’t fully explain why Java disallows this

  ```
  E [] myArray = new E[20];
  ```

  becomes

  ```
  Object [] myArray = new Object[20];
  ```

- One workaround produces a warning

  ```
  E[] myArray = (E []) new Object[20];
  ```

Generics Summary

- Allows us to write code that doesn’t need to specify types
- but requires clients to specify and stick to types
- Provides programmers more representation power than just inheritance
- but not so much freedom as reflection

Reading

- Horstmann Ch. 7.7
- http://java.sun.com/docs/books/tutorial/java/generics/index.html