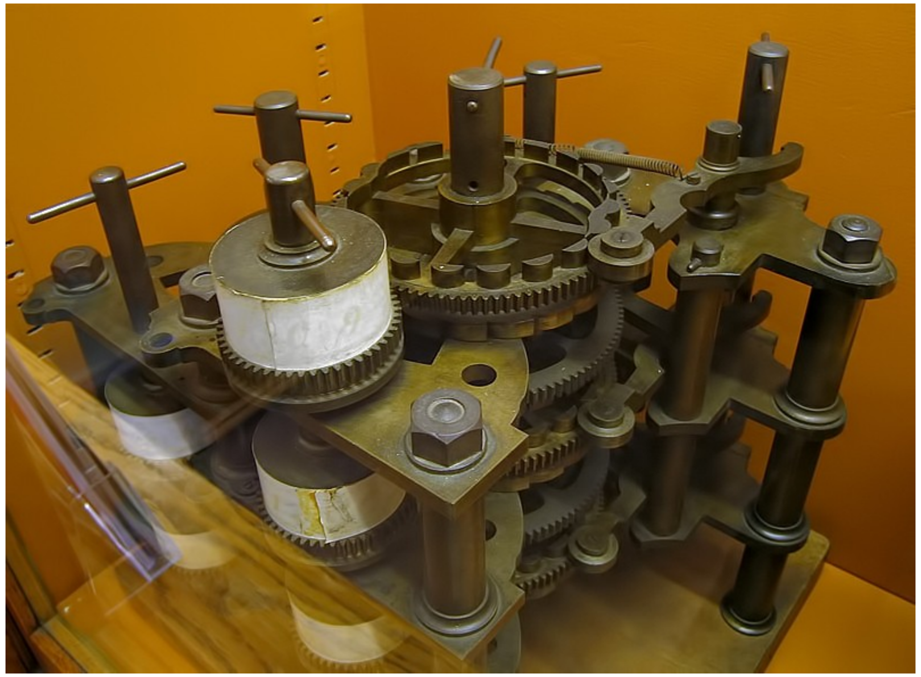
Defensive Programming

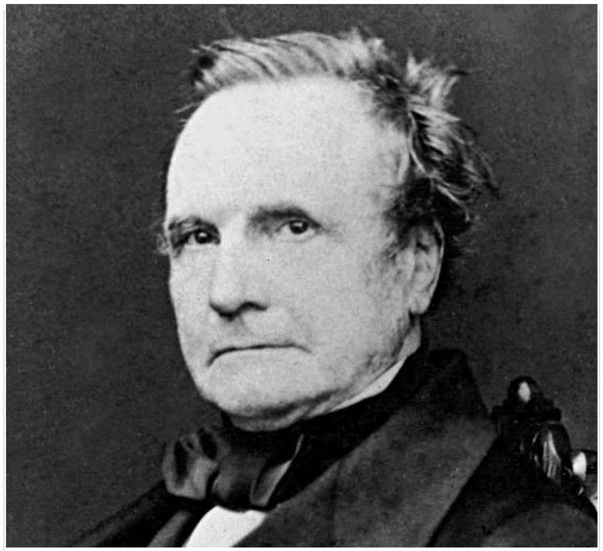
October 10, 2017

Byung-Gon Chun

(Slide credits: George Candea, EPFL)

Garbage In, Garbage Out

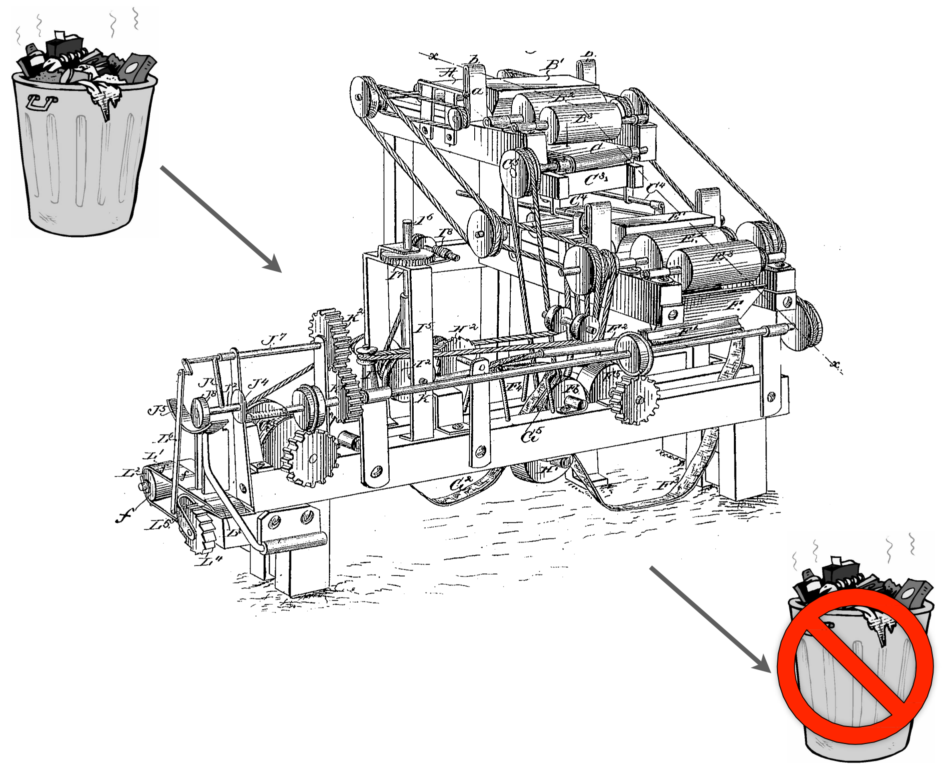

Babbage Difference Engine



On two occasions I have been asked, "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" ... I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.

Charles Babbage
"Passages from the life of a philosopher" (1864)

Our Goal: Garbage In, Non-garbage Out



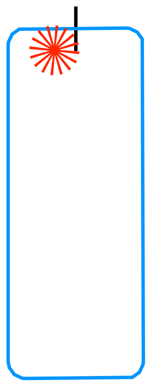
- Sources of garbage
  - Uncontrollable external sources
  - Method parameters
  - Corrupt state
- Options for dealing with garbage:
  - Garbage in, nothing out
  - Garbage in, error message out
  - Turn garbage input into clean input

Dealing with Invalid Inputs

```
/**
 *Returns a BigInteger whose value is (this mod m). This method
 *differs from the remainder method in that it always returns a
 *nonnegative BigInteger.
 *
 *@param   m the modulus, which must be positive.
 *@return this mod m.
 *@throws IllegalArgumentException if m <= 0.
 */
public BigInteger mod(BigInteger m) {
   if (m.signum() <= 0) {
      throw new IllegalArgumentException("Modulus not positive");
   }

   // …
}
```

- Check inputs for validity
  - Can check inputs directly, or via auditErrors()
- Things to check
  - Reference is not null
  - Input params values are within valid range
  - Stream status
  - File access type: read, write, both
- Throw exception if bad input
  - Document pre-conditions of a method "contract"

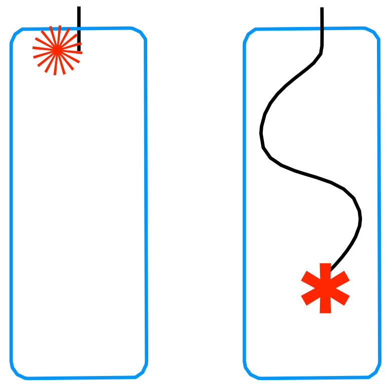
Things That Can Go Wrong



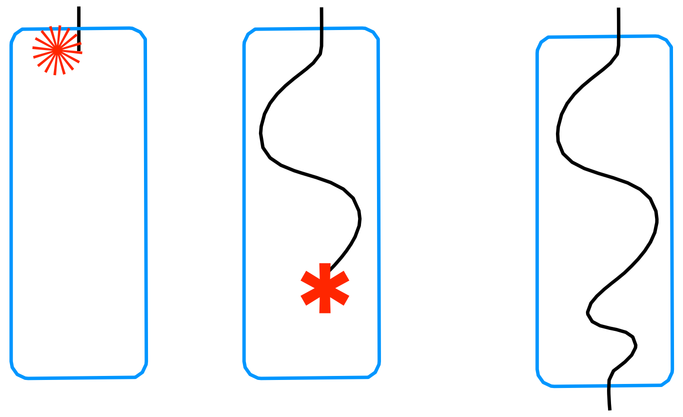
- No check => garbage out

Things That Can Go Wrong

- No check => garbage out
  - Fail with confusing exception later

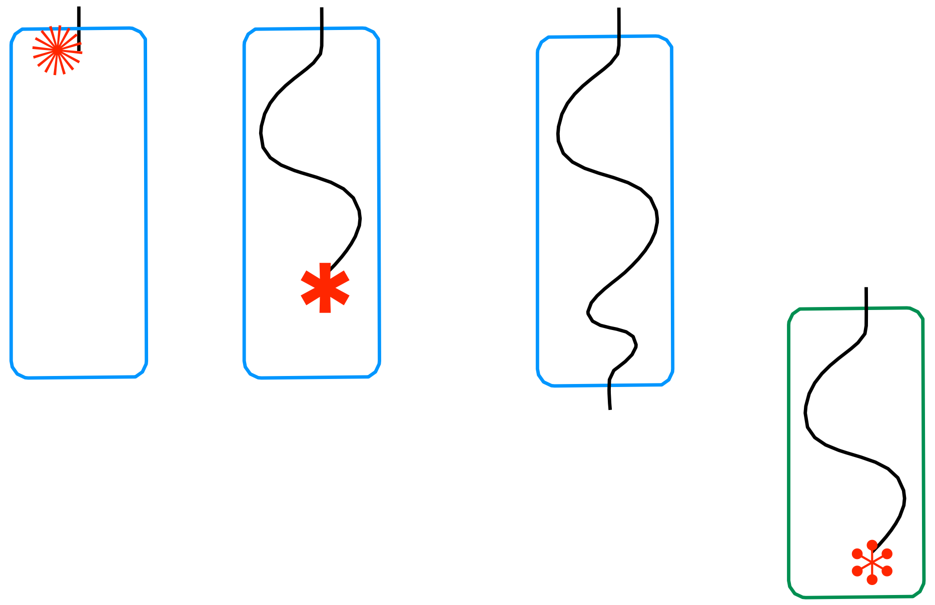


Things That Can Go Wrong



- No check => garbage out
  - Fail with confusing exception later
  - Silently compute the wrong value

Things That Can Go Wrong



- No check => garbage out
  - Fail with confusing exception later
  - Silently compute the wrong value
  - Return normally but compromise some other obj

Things That Can Go Wrong

```
try {
    int i = 0;
    while (true) {
        elements[i++].operation();
    }
} catch (ArrayIndexOutOfBoundsException e) { }
```

```
for (Element el : elements) {
    el.operation();
}
```

- No check => garbage out
  - Fail with confusing exception later
  - Silently compute the wrong value
  - Return normally but compromise some other obj
- Exceptions <= exceptional situations
  - Do not abuse the exception mechanism

Exceptions To The Rule

```java
private void sortList(List<Object> objects) {
   // ...
   Collections.sort(objects, new MyComparator());
}
```

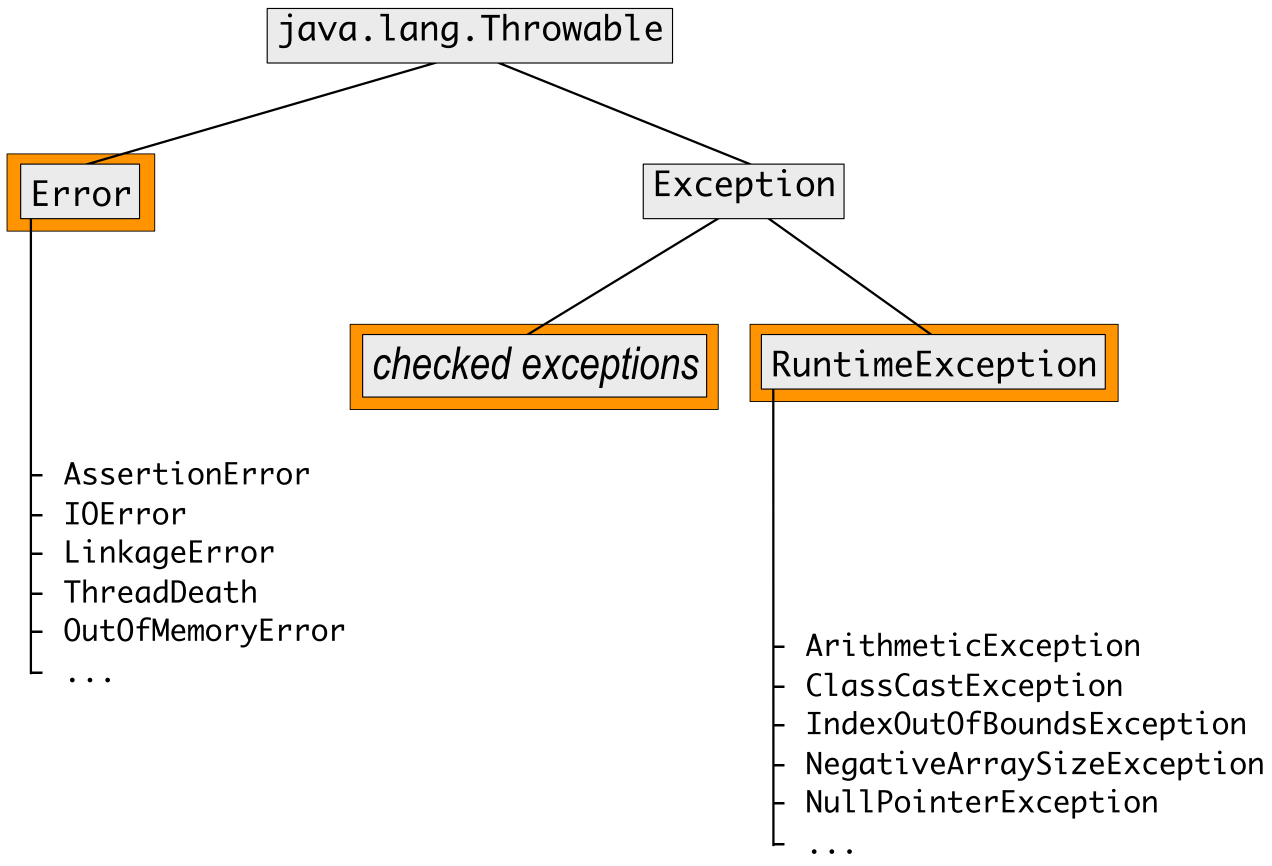
- Avoid checking when…
  - Validity check is too expensive/impractical and it is implicitly done anyway
  - Might need to wrap/translate exception

Preserve Abstraction

```
User me() throws NotLoggedInException {
    // ...
}

User me() throws NotLoggedInException, IOException {
    // ...
}

User me() throws NotLoggedInException, UserDBException {
    // ...
}

User search(String keyword) throws NotFoundException {
    // ...
}
```

- Throw at right level of abstraction
- Aim for informative exceptions
  - Include the context of the condition
  - Put yourself in the shoes of the catcher
- Is an exception needed?

Choosing The Right Exception



- Checked Exceptions
  - Exceptional but recoverable conditions
  - Require try-catch

Fixing Invalid Data

```
Scanner input = new Scanner(System.in);
System.out.println("Enter a lowercase vowel");
while (!input.hasNext("[aeiou]")) {
   System.out.println("Not a vowel; skipping");
   input.next();
}
processVowel(input.next());
```

- Ways to replace/fix invalid data
  - Use the previously used value
  - Use a neutral value
  - Use the next valid entry / element
  - Find closest legal value
- The key trade-off…
  - Cost of throwing exception
    vs.
    cost of getting the input wrong
    x probability of it being wrong

What Are Known Truths?

days in year $\overset{?}{\geq}$ 365

$0 \overset{?}{\geq} \text{seconds} \overset{?}{\geq} 59$

```
int *p,*r;
...some code
...more code
*p = 2;
*r = 3;
assert(*p + *r == 5);
```

```
assert(x+1 > x)
```

What Are Known Truths?

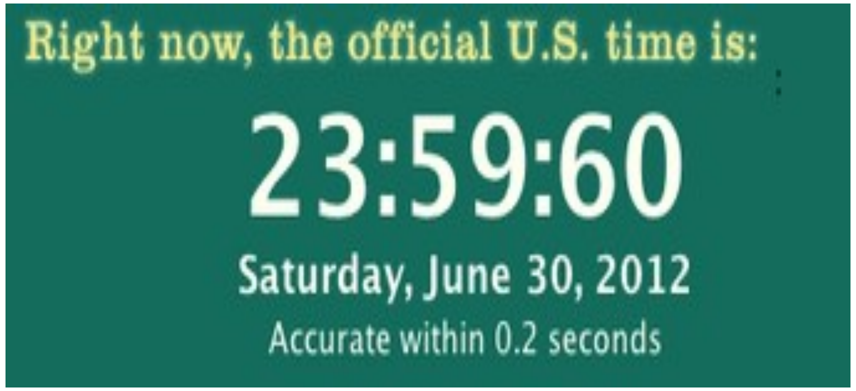
days in year $\overset{?}{\geq}$ 365

days in 1752 = 354

```
int *p,*r;
p = r;
*p = 2;
*r = 3;
assert(*p + *r == 5);
```

0 $\overset{?}{\geq}$ seconds $\overset{?}{\geq}$ 59

... leap seconds


Right now, the official U.S. time is:

23:59:60

Saturday, June 30, 2012

Accurate within 0.2 seconds

```
x=Integer.MAX_VALUE
assert(x+1 > x)
```

Assertions Check Assumptions

```
assert invariant: details
```

- Checks for “impossible” conditions
- Catch bugs during development
  - Mismatched interface assumptions
  - Errors caused by modified code
- Assertions serve as documentation
  - Insurance against future code evolution
- Sanity checks for your program
  - Check parameters of non-public methods
  - Verify code invariants
  - Fulfills a subset of the audit methods’ role
- Java disables assertions by default
  - Use java –ea to enable

Assertions Check Assumptions

```
public class HashMap<K, V>
extends AbstractMap<K, V>
implements Map<K, V>, Cloneable, Serializable {
    // ...
    public HashMap(int initialCapacity, float loadFactor) {
      if (initialCapacity < 0) {
         throw new IllegalArgumentException("Illegal initial capacity: "
               + initialCapacity);
      }
      if (loadFactor <= 0 || Float.isNaN(loadFactor)) {
         throw new IllegalArgumentException("Illegal load factor: "
               + loadFactor);
      }
      // ...
   }
   // ...

   void resize(int newCapacity) {
      assert newCapacity>table.length || table.length==MAXIMUM_CAPACITY;
      // ...
   }
```

Code Invariants

```
public synchronized boolean equals(Object o) {
  // ...
  try {
    Iterator<Map.Entry<K,V>> i = entrySet().iterator();
    loopInvariant must be true here
    while (i.hasNext()) { Map.Entry<K,V> e =
      i.next();
      K key = e.getKey();
      V value = e.getValue();
      if (value == null) {
        if (!(t.get(key)==null && t.containsKey(key))) {
          return false;
        }
      } else {
        if (!value.equals(t.get(key))) {
          return false;
        }
      }
      loopInvariant must be true here
    }
    loopInvariant ∧ !i.hasNext() must be true here
  } catch (ClassCastException unused){
    return false;
  } catch (NullPointerException unused) {
    return false;
  }
```

- "Invariant" means "always true"
  - Property that is purported to always hold
  - Generally restricted to a certain portion of code
  - Examples: loop invariant, class invariant
- Use asserts to enforce invariants

Code Invariants

```
enum Suit {
  CLUBS, DIAMONDS, HEARTS, SPADES;
}
switch(suit) {
case CLUBS:
     // ...
     break;
case DIAMONDS:
     // ...
     break;
case HEARTS:
// ...// ...
     break;
case SPADES:
     // ...
}
```

- “Invariant” means “always true”
  - Property that is purported to always hold
  - Generally restricted to a certain portion of code
  - Examples: loop invariant, class invariant
- Use asserts to enforce invariants
- Use asserts to catch the impossible
  - E.g., empty default statements

Code Invariants

```
enum Suit {
  CLUBS, DIAMONDS, HEARTS, SPADES;
}
// ...
switch(suit) {
 case CLUBS:
      // ...
      break;
 case DIAMONDS:
      // ...
      break;
 case HEARTS:
      // ...
      break;
 case SPADES:
      // ...
      break;
 default:
      throw new AssertionError(suit);
}
 // ...
```

- “Invariant” means “always true”
  - Property that is purported to always hold
  - Generally restricted to a certain portion of code
  - Examples: loop invariant, class invariant
- Use asserts to enforce invariants
- Use asserts to catch the impossible
  - E.g., empty default statements

Defensive Copying

```
public final class Period {
    private final Date start;
    private final Date end;

     public Period(Date dateStart, Date dateEnd) {
        if (dateStart.compareTo(dateEnd) > 0) {
            throw new IllegalArgumentException(dateStart + " after " + dateEnd);
        }
        this.start = dateStart;
        this.end = dateEnd;
    }

     public Date start() {
        return start;
    }
     public Date end() {
        return end;
    }

    // ...
}
```

Defensive Copying

```java
public final class Period {
    private final Date start;
    private final Date end;

     public Period(Date dateStart, Date dateEnd) {
        if (dateStart.compareTo(dateEnd) > 0) {
            throw new IllegalArgumentException(dateStart + " after " + dateEnd);
        }
        this.start = dateStart;
        this.end = dateEnd;
    }

     public Date start() {
        return start;
    }
     public Date end() {
        return end;
    }

    // ...
}
```

```java
Date s = new Date();
Date e = new Date();
Period p = new Period(s, e);
e.setYear(78); // Modifies internals of p
```

Defensive Copying

```
public final class Period {
    private final Date start;
    private final Date end;

    public Period(Date dateStart, Date dateEnd) {
        this.start = new Date(dateStart.getTime());
        this.end = new Date(dateEnd.getTime());

        if (dateStart.compareTo(dateEnd) > 0) {
            throw new IllegalArgumentException(dateStart + " after " + dateEnd);
        }
    }

    public Date start() {
        return start;
    }
    public Date end() {
        return end;
    }

    // ...
}
```

```
Date s = new Date();
Date e = new Date();
Period p = new Period(s, e);
e.setYear(78); // Modifies internals of p
```

Defensive Copying

```
public final class Period {
   private final Date start;
   private final Date end;

    public Period(Date dateStart, Date dateEnd) {
       this.start = new Date(dateStart.getTime());
       this.end = new Date(dateEnd.getTime());

       if (dateStart.compareTo(dateEnd) > 0) {
          throw new IllegalArgumentException(dateStart + " after " + dateEnd);
       }
   }

    public Date start() {
       return start;
   }
    public Date end() {
       return end;
   }

   // ...
}
```

```
Date s = new Date();
Date e = new Date();
Period p = new Period(s, e);
```

~~e.setYear(78); // Modifies internals of p~~

Defensive Copying

```
public final class Period {
   private final Date start;
   private final Date end;

    public Period(Date dateStart, Date dateEnd) {
       this.start = new Date(dateStart.getTime());
       this.end = new Date(dateEnd.getTime());

       if (dateStart.compareTo(dateEnd) > 0) {
          throw new IllegalArgumentException(dateStart + " after " + dateEnd);
       }
   }

    public Date start() {
       return start;
   }
    public Date end() {
       return end;
   }

   // ...
}
```

```
Date s = new Date();
Date e = new Date();
Period p = new Period(s, e);
```
~~e.setYear(78); // Modifies internals of p~~
```
p.end().setYear(78);
```

Defensive Copying

```
public final class Period {
    private final Date start;
    private final Date end;

    public Period(Date dateStart, Date dateEnd) {
        this.start = new Date(dateStart.getTime());
        this.end = new Date(dateEnd.getTime());

        if (dateStart.compareTo(dateEnd) > 0) {
            throw new IllegalArgumentException(dateStart + " after " + dateEnd);
        }
    }

    public Date start() {
        return (Date) start.clone();
    }
    public Date end() {
        return (Date) end.clone();
    }

    // ...
}
```

```
Date s = new Date();
Date e = new Date();
Period p = new Period(s, e);
```
~~e.setYear(78); // Modifies internals of p~~
```
p.end().setYear(78);
```

Defensive Copying

```
public final class Period {
    private final Date start;
    private final Date end;

    public Period(Date dateStart, Date dateEnd) {
        this.start = new Date(dateStart.getTime());
        this.end = new Date(dateEnd.getTime());

        if (dateStart.compareTo(dateEnd) > 0) {
            throw new IllegalArgumentException(dateStart + " after " + dateEnd);
        }
    }

    public Date start() {
        return (Date) start.clone();
    }
    public Date end() {
        return (Date) end.clone();
    }

    // ...
}
```

```
Date s = new Date();
Date e = new Date();
Period p = new Period(s, e);
```

~~e.setYear(78); // Modifies internals of p~~

~~p.end().setYear(78);~~

Defensive Programming

- Check inputs
  - Can use exceptions for public methods, assertions for non-public ones
  - Discard bad inputs, repair bad inputs
- Document assumptions
  - Cannot control outside world, but can be explicit about what we assume about it
- Check code invariants
- Employ defensive copying