Input Validation

Input Errors

- Example 1: $1 Billion typing error:
  - In 2005, a Japanese securities trader mistakenly sold 600,000 shares of stock at 1 yen each , rather than 600,000 yen each.
- Example 2: $100,000 typing error:
  - A Norwegian woman mistyped her account number by adding an extra digit to her 11-digit account number. The system discarded the extra digit, and transferred $100,000 to the (incorrect) account.
- Both of these errors were preventable by simple input validation checks!
  - Example 1: Check price >= the minimum price per share
  - Example 2: Check the account number has the correct number of digits

Input Errors cause Security Vulnerabilities

- Input errors can be caused by
  - accidental mistakes by trusted users
  - Malicious users looking to take advantage of flaws in the system
    - malicious user: one who intentionally crafts input data to cause programs to run unauthorized commands
    - Discuss: How can a malicious person take advantage of the input errors from the previous slide?

Malicious Input Error Attacks

- Credit cards stolen:
  - In Feb 2002, Jeremiah Jacks discovered that at Guess.com a properly-crafted URL allowed anyone to pull down 200,000+ names, credit card numbers and expiration dates in the site's customer database.
  - Known as a SQL-Injection attack
  - The attack is carried out for example by entering in a SQL command into a search box

Summary

- Programs often use external data
  - User input, file, database, network
- All external data that can enter your program can be a potential source of problems.
- Using external data without validation can make your system susceptible to security vulnerabilities.

What to do if input has errors?

- When input errors are detected the program should immediately reject the request.
  - Do not attempt to interpret erroneous input into a correct one. Why?
    - Malicious user can craft input in a way so that the corrected version is an attack
- This is called the deny-by-default design principal
  - anything not explicitly permitted is forbidden.

Common ways to validate input data

1. **Range check (reasonableness check)** - numbers checked to ensure they are within a range of possible values, e.g.,
   - the value for month should lie between 1 and 12.
   - Stocks cannot be sold for less than 1 yen
2. **Length check:** ensure input is of appropriate length, e.g.,
   - US telephone number has 10 digits.
   - Bank account numbers are 11 digits long
3. **Type check:** input should be checked to ensure it is the data type expected, e.g.,
   - age must be integer.
4. **Format check** – Check that the data is in a specified format (template),
   - e.g., dates might be required to be in the format DD/MM/YYYY.
5. **Arithmetic Errors:** variables are checked for values that might cause problems such as
   - division by zero or integer overflow.

Input Validation on Split Bill Program

(SplitBill_v1.java)

Where is the External Data?

```
public void computeBill()
{
   Scanner scan = new Scanner(System.in);

   System.out.print("Enter the total bill:");
   double bill = scan.nextDouble();

   System.out.print("Enter the number of people in your party:");
   int people = scan.nextInt();

   System.out.println("Each person owes $" + bill/people);

   scan.close();
}
```

Common ways to validate input data

1. **Range check (reasonableness check)** - numbers checked to ensure they are within a range of possible values
   - 0 < Bill < 2000 0 < people < 10
2. **Length check:** ensure input is of appropriate length
   - Does not apply in SplitBill program
3. **Type check:** input should be checked to ensure it is the data type expected
   - Bill should be a double, people should be an int
4. **Format check** – Check that the data is in a specified format (template)
   - Does not apply in SplitBill program
5. **Arithmetic Errors:** variables are checked for values that might cause problems such as
   - People cannot be 0

Range Check Example

(SplitBill_v2.java)

- if bill is < 0 or > 2000 show error message

```
public void computeBill()
{
   Scanner scan = new Scanner(System.in);

   System.out.print("Enter the total bill:");
   double bill = scan.nextDouble();

   if (bill < 0 || bill > 2000)
      System.out.println("Error! Bill must be between 0 and $2000");

   System.out.print("Enter the number of people in your party:");
   int people = scan.nextInt();

   System.out.println("Each person owes $" + bill/people);

   scan.close();
}
```

What is the flaw in this solution?

Answer: Processing continues even when bill is out of range; does not follow deny by default

Range Check Example

(SplitBill_v2Better.java)

- Verifies that bill is between 0 and 2000
  - If not, display error and terminate program

```
public void computeBill()
{
   Scanner scan = new Scanner(System.in);

   System.out.print("Enter the total bill:");
   double bill = scan.nextDouble();

   if (bill < 0 || bill > 2000)
      System.out.println("Error! Bill must be between 0 and $2000");
   else
   {
      System.out.print("Enter the number of people in your party:");
      int people = scan.nextInt();

      System.out.println("Each person owes $" + bill/people);
   }

   scan.close();
}
```

Range Check Exercise

- Update code below to do range check so that people is between 0 and 10

```
public void computeBill()
{
   Scanner scan = new Scanner(System.in);

   System.out.print("Enter the total bill:");
   double bill = scan.nextDouble();

   if (bill < 0 || bill > 2000)
      System.out.println("Error! Bill must be between 0 and $2000");
   else
   {
      System.out.print("Enter the number of people in your party:");
      int people = scan.nextInt();

      System.out.println("Each person owes $" + bill/people);
   }

   scan.close();
}
```

Range Check Exercise Solution

```
public void computeBill()
{
   Scanner scan = new Scanner(System.in);

   System.out.print("Enter the total bill:");
   double bill = scan.nextDouble();

   if (bill < 0 || bill > 2000)
      System.out.println("Error! Bill must be between 0 and $2000");
   else
   {
      System.out.print("Enter the number of people in your party:");
      int people = scan.nextInt();
      if (people < 1 || people > 10)
         System.out.println("Error! Party must be between 1 and 10");
      else
         System.out.println("Each person owes $" + bill/people);
   }

   scan.close();
}
```

Improving current Solution

- Some unattractive qualities of current solution:
  - Multiple if/else statements makes code less readable, cumbersome to update, error prone
    - Imagine if there were 10 input values instead of just 2
    - What if some ranges needed to be updated?
  - If user makes a mistake on any input they must start over from the beginning of the program
    - Imagine making an error on the 10th input value – need to reenter all values again.

Improve current Solution: Use methods

- Use a separate method to read and validate each type of input value

```
private double getBill(Scanner scan)
{
   System.out.print("Enter the total bill:");
   double bill = scan.nextDouble();
   if (bill < 0 || bill > 2000)
   {
      System.out.println("Error! Bill must be between 0 and $2000");
      bill = -1;
   }
   return bill;
}
```

Improve current solution: Using methods

- Benefit: separation of concerns
  - Computation in one method, input validation in another

```
private void computeBill()
{
   Scanner scan = new Scanner(System.in);

   double bill = getBill(scan);
   double people = getPeople(scan);

   if (bill!= -1 && people != -1)
      System.out.println("Each person owes $" + bill/people);

   scan.close();
}
```

Improve current Solution: Use methods

- Use the example of getBill() to implement getPeople().
  - Return a valid value for people (between 0 and 10) or -1 if value was not in range

```
private double getBill(Scanner scan)
{
   System.out.print("Enter the total bill:");
   double bill = scan.nextDouble();
   if (bill < 0 || bill > 2000)
   {
      System.out.println("Error! Bill must be between 0 and $2000");
      bill = -1;
   }
   return bill;
}
```

Improve current Solution: Use methods

- Use the example of getBill() to implement getPeople().
  - Return a valid value for people (between 0 and 10) or -1 if value was not in range
  - See SplitBill_v3.java for full solution

```
private double getPeople(Scanner scan)
{
   System.out.print("Enter the number of people in your party:");
   int people = scan.nextInt();
   if (people < 1 || people > 10)
   {
      System.out.println("Error! Party must be between 1 and 10");
      people = -1;
   }
   return people;
}
```

Common ways to validate input data

- ✓ **Range check (reasonableness check)** - numbers checked to ensure they are within a range of possible values
  - 0 < Bill < 2000 0 < people < 10
- **Length check:** ensure input is of appropriate length
  - Does not apply in SplitBill program
- **Type check:** input should be checked to ensure it is the data type expected
  - Bill should be a double, people should be an int
- **Format check** – Check that the data is in a specified format (template)
  - Does not apply in SplitBill program
- ✓ **Arithmetic Errors:** variables are checked for values that might cause problems such as
  - People cannot be 0

Type Checking

- Prevent user from entering incorrect type of data e.g.,
  - "hello" for bill or "4.5" for people

```
Enter the total bill:
hello
Exception in thread "main" java.util.InputMismatchException
at java.util.Scanner.throwFor(Unknown Source)
at java.util.Scanner.next(Unknown Source)
at java.util.Scanner.nextDouble(Unknown Source)
at SplitBill.getBill(SplitBill.java:19)
at SplitBill.main(SplitBill.java:7)
```

```
Enter the total bill:
30
Enter the number of people in your party:
4.5
Exception in thread "main" java.util.InputMismatchException
at java.util.Scanner.throwFor(Unknown Source)
at java.util.Scanner.next(Unknown Source)
at java.util.Scanner.nextInt(Unknown Source)
at java.util.Scanner.nextInt(Unknown Source)
at SplitBill.getPeople(SplitBill.java:33)
at SplitBill.main(SplitBill.java:8)
```

Type Checking

- Before reading input using nextDouble or nextInt, make sure there is a double or int to read
- Useful Scanner methods for this purpose
  - hasNextInt(), hasNextDouble()
- If next input is not of the right type skip over it using next() method

Type Checking

```
private double getBill(Scanner scan)
{
   double bill;
   System.out.print("Enter the total bill:");
   if(scan.hasNextDouble())
      bill = scan.nextDouble();
   else
   {
      scan.next(); //skip over incorrect input
      bill = -1;
   }

   if (bill < 0 || bill > 2000)
   {
      System.out.println("Error! Bill must be a number between 0 and $2000");
      bill = -1;
   }
   return bill;
}
```

Exercise

- Update the getPeople method to use type checking using hasNextInt()

```
private double getPeople(Scanner scan)
{
   System.out.print("Enter the number of people in your party:");
   int people = scan.nextInt();
   if (people < 1 || people > 10)
   {
      System.out.println("Error! Party must be between 1 and 10");
      people = -1;
   }
   return people;
}
```

Exercise Solution

- Update the getPeople method to use type checking using hasNextInt()

```
private double getPeople(Scanner scan)
{
   int people;
   System.out.print("Enter the number of people in your party:");
   if(scan.hasNextInt())
      people = scan.nextInt();
   else
   {
      scan.next(); //skip over incorrect input
      people = -1;
   }

   if (people < 1 || people > 10)
   {
      System.out.println("Error! Party must be an integer between 1 and 10");
      people = -1;
   }
   return people;
}
```

Common ways to validate input data

- ✓ **Range check (reasonableness check)** - numbers checked to ensure they are within a range of possible values
  - 0 < Bill < 2000 0 < people < 10
- **Length check:** ensure input is of appropriate length
  - Does not apply in SplitBill program
- ✓ **Type check:** input should be checked to ensure it is the data type expected
  - ✓ Bill should be a double, people should be an int
- **Format check** – Check that the data is in a specified format (template)
  - Does not apply in SplitBill program
- ✓ **Arithmetic Errors:** variables are checked for values that might cause problems such as
  - People cannot be 0